

C.A.R. HOARE

- Profesor în Teoria programării -
Universitatea Oxford

PROCESE SECVENȚIALE , COMUNICANTE

Editura Fundației

Chemarea

IAȘI

1995

681.3
H68

C.A.R. HOARE

– Profesor în Teoria programării –
Universitatea Oxford

Scanned with CamScanner

PROCESE SECVENȚIALE COMUNICANTE

Traducere din limba engleză: ș.l. NICOLAE MARIAN
ș.l. DAN OLARU



675.673

Editura Fundației *Chemarea* I A Ș I
1995

Cuprins

Cuvânt înainte	7
Prefață și aprecieri	9
Glosar de simboluri	15
1 Procese	21
1.1 Introducere	21
1.1.1 Prefix	23
1.1.2 Recursivitatea	25
1.1.3 Alegere	27
1.1.4 Recursivitatea mutuală	31
1.2 Reprezentări grafice	32
1.3 Legi	33
1.4 Implementarea proceselor	36
1.5 Urme	39
1.6 Operații cu urme	41
1.6.1 Concatenarea	41
1.6.2 Restricția	42
1.6.3 Cap și coadă	43
1.6.4 Stea	43
1.6.5 Ordonarea	44
1.6.6 Lungime	45

1.7	Implementarea urmelor	46
1.8	Urmele proceselor	47
1.8.1	<i>Legi</i>	48
1.8.2	<i>Implementare</i>	51
1.8.3	<i>După</i>	51
1.9	Alte operații cu urme	54
1.9.1	<i>Schimbare de simbol</i>	54
1.9.2	<i>Unificare</i>	55
1.9.3	<i>Întrețeserea</i>	55
1.9.4	<i>Indexarea</i>	56
1.9.5	<i>Reversul</i>	56
1.9.6	<i>Selecția</i>	56
1.9.7	<i>Compunerea</i>	57
1.10	Specificații	58
1.10.1	<i>Satisfacerea</i>	59
1.10.2	<i>Reguli</i>	61

2 Concurență 65

2.1	Introducere	66
2.2	Interacțiune	65
2.2.1	<i>Legi</i>	67
2.2.2	<i>Implementare</i>	68
2.2.3	<i>Urme</i>	68
2.3	Concurență	68
2.3.1	<i>Legi</i>	70
2.3.2	<i>Implementare</i>	72
2.3.3	<i>Urme</i>	72
2.4	Reprezentări grafice	73
2.5	Exemplu : Problema mesei celor cinci filozofi	75
2.5.1	<i>Alfabet</i>	75
2.5.2	<i>Comportarea</i>	76
2.5.3	<i>Blocaj</i>	77
2.5.4	<i>Demonstrarea lipsei blocajului</i>	78
2.5.5	<i>Preluare infinită</i>	80
2.6	Schimbare de simbol	80
2.6.1	<i>Legi</i>	84
2.6.2	<i>Etichetarea proceselor</i>	85
2.6.3	<i>Implementare</i>	88
2.6.4	<i>Etichetarea multiplă</i>	89
2.7	Specificații	90

2.8	Teoria matematică a proceselor deterministe	92
2.8.1	<i>Definiții de bază</i>	93
2.8.2	<i>Teoria de punct fix</i>	95
2.8.3	<i>Unicitatea soluțiilor</i>	97

3 Nedeterminism 103

3.1	Introducere	103
3.2	SAU nedeterminist	103
3.2.1	<i>Legi</i>	104
3.2.2	<i>Implementări</i>	106
3.2.3	<i>Urme</i>	108
3.3	Alegere generală	108
3.3.1	<i>Legi</i>	109
3.3.2	<i>Implementare</i>	110
3.3.3	<i>Urme</i>	110
3.4	Refuzuri	110
3.4.1	<i>Legi</i>	112
3.5	Măscare (ascundere)	113
3.5.1	<i>Legi</i>	114
3.5.2	<i>Implementare</i>	118
3.5.3	<i>Urme</i>	119
3.5.4	<i>Reprezentări</i>	119
3.6	Întrețeserea	122
3.6.1	<i>Legi</i>	122
3.6.2	<i>Urme și refuzuri</i>	123
3.7	Specificații	124
3.7.1	<i>Reguli</i>	126
3.8	Divergență	128
3.8.1	<i>Legi</i>	130
3.8.2	<i>Divergențe</i>	130
3.9	Teoria matematică a proceselor nedeterministe	132

4 Comunicația 137

4.1	Introducere	137
4.2	Intrare și ieșire	138
4.2.1	<i>Implementare</i>	143
4.2.2	<i>Specificații</i>	144

4.3	Comunicații	147
4.4	Conducte	156
4.4.1	<i>Legi</i>	159
4.4.2	<i>Implementare</i>	161
4.4.3	<i>Blocaj prin ciclare infinită</i>	161
4.4.4	<i>Specificații</i>	163
4.4.5	<i>Buffere și protocoale</i>	164
4.5	Subordonarea	168
4.5.1	<i>Legi</i>	172
4.5.2	<i>Diagrame de conexiune</i>	174

5 Procese secvențiale 179

5.1	Introducere	179
5.2	Legi	183
5.3	Aspecte matematice	185
5.3.1	<i>Procese deterministe</i>	185
5.3.2	<i>Procese nedeterministe</i>	187
5.3.3	<i>Implementare</i>	188
5.4	Întreruperi	188
5.4.1	<i>Catastrofă</i>	190
5.4.2	<i>Restart</i>	190
5.4.3	<i>Alternarea</i>	191
5.4.4	<i>Puncte de control</i>	192
5.4.5	<i>Puncte de control multiple</i>	193
5.4.6	<i>Implementare</i>	194
5.5	Asignare	195
5.5.1	<i>Legi</i>	197
5.5.2	<i>Specificații</i>	199
5.5.3	<i>Implementare</i>	205

6 Resurse partajate 207

6.1	Introducere	207
6.2	Partajarea prin întrețesere	208
6.3	Partajarea memoriei	213
6.4	Resurse multiple	215
6.5	Sisteme de operare	226
6.6	Planificare	231

7	Discuții	235
7.1	Introducere	235
7.2	Partajarea memorării	235
7.2.1	<i>Multiplicarea firelor de execuție</i>	236
7.2.2	<i>Cobegin ... coend</i>	237
7.2.3	<i>Regiuni critice condiționale</i>	238
7.2.4	<i>Monitoare</i>	240
7.2.5	<i>Monitoare imbricate</i>	242
7.2.6	<i>Ada</i>	244
7.3	Comunicarea	247
7.3.1	<i>Conducte</i>	247
7.3.2	<i>Canale bufferate multiple</i>	248
7.3.3	<i>Multiprocesare funcțională</i>	248
7.3.4	<i>Comunicație nebufferată</i>	250
7.3.5	<i>Procese secvențiale comunicante</i>	251
7.3.6	<i>Occam</i>	252
7.4	Modele matematice	255
7.4.1	<i>Calculul sistemelor comunicante</i>	256
	Bibliografie	263
	Index	265

7	Discuții	235
7.1	Introducere	235
7.2	Partajarea memorării	235
7.2.1	<i>Multiplicarea firelor de execuție</i>	236
7.2.2	Cobegin ... coend	237
7.2.3	<i>Regiuni critice condiționale</i>	238
7.2.4	<i>Monitoare</i>	240
7.2.5	<i>Monitoare imbricate</i>	242
7.2.6	<i>Ada</i>	244
7.3	Comunicarea	247
7.3.1	<i>Conducte</i>	247
7.3.2	<i>Canale bufferate multiple</i>	248
7.3.3	<i>Multiprocesare funcțională</i>	248
7.3.4	<i>Comunicație nebufferată</i>	250
7.3.5	<i>Procese secvențiale comunicante</i>	251
7.3.6	<i>Occam</i>	252
7.4	Modele matematice	255
7.4.1	<i>Calculul sistemelor comunicante</i>	256
	Bibliografie	263
	Index	265

Cuvânt înainte

Din mai multe motive, aceasta este o carte așteptată de toți cei care știau că se pregătește. A spune acum că răbdarea lor a fost pe deplin răsplătită ar fi o litotă.

Un simplu motiv face ca aceasta să fie prima carte a lui Tony Hoare. Mulți îl cunosc din prelegerile pe care le-a ținut neobosit peste tot și mai mulți îl știau ca autor atent și pedant al unui număr de articole (de o mare varietate!) care au devenit clasice aproape înainte de a se usca cerneala. Dar o carte este altceva : aici autorul se poate exprima pe sine fără limitările obișnuite de timp și spațiu. De asemenea i se dă posibilitatea unei relevări proprii mai intime, acoperirii unei topici de o dimensiune mărită, ocazie pe care Tony Hoare a fructificat-o mai bine decât ne așteptam.

Un motiv mai solid transpare însă din conținutul cărții. Când concurența și-a făcut loc în conceptele de calcul acum un sfert de secol, a provocat o confuzie fără sfârșit, parțial datorată circumstanțelor diferite de unde provenea, parțial datorată introducerii nedeterminismului în acea perioadă. Descălcirea acestei confuzii a necesitat munca încordată a unui profesionist devotat, care cu puțin noroc, putea clarifica situația. Tony Hoare și-a dedicat o mare parte din activitatea științifică acestei provocări și avem toate motivele să-i mulțumim.

Cel mai puternic motiv, totuși, a fost subtil resimțit de acei care au recenzat manunscrisul, ce aruncă o nouă, surprinzătoare și în același timp clară lumină asupra a ceea ce știința calculatoarelor poate – sau va – fi. A spune sau simți că principalul obiectiv al unui savant din lumea calculatoarelor este de a nu crea confuzie prin propriile descoperiri este un aspect și cu totul altceva este de a descoperi și arăta cum eleganța tangibilă și aproape explicită a câtorva legi matematice călăuzitoare poate realiza o teorie sublimă. De aceea noi, cititorii recunoscători, vom alege după gust roadele inteligenței științifice, tenacității și agilității conceptuale ale lui C.A.R. Hoare.

Edsger W. Dijkstra

Prefață

Aceasta este o carte pentru informaticienii dornici de o mai bună înțelegere și perfecționare într-o profesie intelectuală dintre cele mai dinamice. Se adresează întâi unei curiozități firești provenită din noua tratare a unui domeniu cunoscut, fiind ilustrată de exemple selectate dintr-o gamă mare de aplicații : automate de vânzare, jocuri, sisteme de operare. Totul se bazează pe o teorie matematică descrisă sistematic prin legi algebrice. Obiectivul principal este de a introduce un punct de vedere ce permite cititorului să vadă problemele actuale și viitoare într-o lumină nouă în care rezolvările sunt mai sigure și eficiente sau, chiar mai mult, pot fi evitate neajunsurile.

Cea mai importantă aplicație a noilor concepte este specificarea, modelarea și implementare sistemelor care interacționează continuu cu mediul. Ideea de bază este posibilitatea descompunerii fără dificultate a acestor sisteme în subsisteme care operează concurent și interacționează între ele și mediu. Compunerea paralelă a subsistemelor este la fel de simplă ca și compunerea secvențială a instrucțiunilor în limbajele de programare convenționale.

Această privire interioară aduce și beneficii practice. Întâi evită multe din problemele paralelismului în programare – interferență, excludere mutuală, întreruperi, fire de execuție, semafoare etc. Apoi include drept cazuri particulare multe din ideile de structurare inovatoare care au fost dezvoltate prin cercetări recente în domeniul limbajelor de programare și metodologiei de programare – monitor, clasă, modul, pachet, regiune critică, plic, formă și chiar umila subrutină. În fine, asigură un fundament matematic sigur pentru evitarea erorilor ca divergență, blocaj, blocaj prin ciclare infinită și pentru realizarea unei corectitudini demonstrabile în modelarea și implementarea sistemelor informatice.

Am încercat din răspuțeri să prezint ideile într-o ordine logică și psihologică strictă, pornind de la operatori de bază simpli și avansând către aplicații elaborate. Cititorii vor avea sigur un grad diferit de interes, pornind de la cei care vor citi cartea din scoarță în scoarță și până la cei interesați numai de anumite probleme. De aceea fiecare capitol al cărții este structurat pentru a permite o judicioasă selecție.

1. Fiecare idee nouă este mai întâi descrisă și ilustrată de un număr de exemple simple, care probabil vor fi de ajutor pentru toți cititorii.

2. Legile algebrice care descriu proprietățile esențiale ale diversilor operatori vor interesa mai mult pe cei ce sunt atrași de eleganța matematică. Ele vor fi de ajutor și celor care doresc o optimizare a modelării prin utilizarea unor transformări echivalente cu păstrarea corectitudinii.
3. Implementările propuse sunt deosebite prin aceea că utilizează un subset minimal, funcțional, simplu al limbajului de programare LISP. Aceasta permite celor familiarizați cu LISP-ul să realizeze exerciții și demonstrații.
4. Definițiile urmelor și specificațiilor sunt de interes pentru analiștii de sistem, care trebuie să exprime nevoile clientului înainte de a considera o implementare. Ele sunt de folos programatorilor experimentați, ce proiectează sisteme prin împărțirea în subsisteme cu specificarea clară a interfețelor.
5. Demonstrațiile interesează pe cei ce implementează, care trebuie să producă programe fiabile din specificațiile date, la termene fixate și cu prețuri de cost fixate.
6. În fine, teoria matematică dă o definiție riguroasă conceptului de proces și operatorilor, în contextul construirii proceselor. Aceste definiții sunt baza pentru legile algebrice, implementări și demonstrații.

Cititorul poate omite consecvent sau nu parcurgerea unora dintre aceste probleme, de un interes considerat mai scăzut sau cu un grad mai mare de dificultate.

Succesiunea capitolelor în carte a fost astfel organizată pentru a permite o judicioasă parcurgere, selecție sau rearanjare. Primele paragrafe din cap. 1 și 2 sunt o introducere necesară pentru toți cititorii. Celelalte paragrafe pot fi trecute în grabă sau amânate pentru o a doua lectură. Cap. 3, 4, 5 sunt independente unul de altul și pot fi începute în orice combinație corespunzător interesului și înclinației cititorului. Astfel, dacă se întâmpină o anumită dificultate de înțelegere într-o etapă, este preferabil să se continue cu paragraful următor sau cu capitolul următor, deoarece în mod normal materialul omis nu va fi necesar imediat. Când totuși o asemenea necesitate există, va fi în text o referință explicită înapoi, ce poate fi urmată sau nu. Sper ca totul în carte să fie interesant și să justifice efortul de a o citi. Ordinea parcurgerii ține de preferințele fiecăruia.

Exemplele alese pentru a ilustra ideile din carte vor părea foarte simple. Aceasta este și intenția. Primele exemple selectate pentru a ilustra o nouă idee trebuie să fie foarte simple pentru a nu umbri ideea prin complexitatea sau neobișnuitul lor. Simplitatea soluțiilor acestor exemple trebuie să se datoreze puterii conceptelor folosite și eleganței notațiilor.

Cu toate acestea, fiecare cititor are problemele sale, uneori dureroase, mai complexe, mai importante decât exemplele alese. Aceste probleme par a nu putea fi rezolvate de vreo teorie matematică. Nu vă lăsați pradă disperării

și enervării și acceptați provocarea aplicării metodelor din această carte problemelor D-voastră. Începeți cu o variantă foarte simplificată a problemelor și adăugați mereu câte ceva până ajungeți la complexitatea cerută. Este surprinzător cum modelul simplificat poate fi o bază de pornire iar detaliile se pot suprapune fără afectarea siguranței. Surpriza finală este că unele detalii pot fi dovedite ca nefiind strict necesare, caz în care teoria își relevă întreaga sa putere.

După cum știm, notațiile sunt o permanentă problemă. Astfel, orice student pus să învețe limba rusă întâmpină dificultăți cu alfabetul chirilic, în special datorită pronunției. Ca o consolare, aceasta este o ultimă problemă ce s-ar întâlni aici. După învățarea simbolurilor se învață gramatica, vocabularul, idioamele, stilul și apoi se poate trece la fluentizarea exprimării ideilor. Toate acestea se fac însă treptat, cu exercițiu, și nu pot fi grăbite. Același lucru este și cu teoria matematică. Simbolurile apar la început ca o piedică, problema reală fiind de a învăța semnificația și proprietățile lor precum și posibilitățile de manipulare, pentru obținerea unei dexterități în exprimarea noilor soluții, demonstrații la diverse probleme. În fine, prin eleganța matematică și stilul folosit, simbolurile vor fi transparente, limpezindu-se și înțelesul lor. Avantajul covârșitor al matematicii este simplitatea regulilor față de limbajul natural și vocabularul redus. De asemenea, o soluție poate fi găsită prin deducție logică, prin consultarea unor cărți sau experți.

Din această cauză atât matematica cât și programarea sunt o permanentă provocare, nu totdeauna ușoară. Chiar matematicienii întâmpină dificultăți cu domenii noi. Teoria proceselor comunicante este un astfel de nou domeniu, iar programatorii care îl abordează nu trebuie să aibă nici un handicap față de matematicienii, din contră, au avantajul de a-și pune cunoștințele în practică. Materialul acestei cărți provine din prezentări în simpozioane, prelegeri și cursuri academice. El a fost proiectat la început ca un curs universitar de un semestru în inginerie software, postgraduate, cu toate că cea mai mare parte poate fi prezentat în a doua jumătate a anului întâi, sau prima parte a anului doi, undergraduate, secția calculatoare. Cerințele minime sunt de algebră superioară de liceu, concepte din teoria mulțimilor și calculul predicatelor. Cartea este adecvată și pentru un curs intensiv de o săptămână pentru programatori experimentați. Într-un astfel de curs, lectorul ar trebui să se concentreze pe exemple și definiții, lăsând la o parte aparatul matematic pentru studiul individual. Este posibil și prezentare sub formă de curs numai a primelor două capitole, sau într-o singură prelegere a problemei celor cinci filozofi.

Nu este deloc ușor de ținut cursuri și seminarii în domeniul proceselor secvențiale comunicante. Exemplele ce trebuie folosite constituie mereu o problemă, dacă punem la socoteală numai gradul de percepție al participanților, sensibilitatea lor. Având în vedere eleganța abstractă a formulelor matematice, unele concepte prezentate în carte au ceva asemănător unor 'fugi' de S. Bach

Sumar

Cap. 1 introduce conceptele de bază ale proceselor și abstractizează matematic interacțiunile dintre sistem și mediu. Se arată cum familiara tehnică a recursivității poate fi folosită pentru a descrie procese de o durată variabilă sau chiar infinită. Conceptele sunt explicate întâi prin exemple și grafic. Explicația completă o oferă legile algebrice iar implementarea se face în limbajul funcțional LISP. A doua parte a capitolului explică cum comportarea unui proces poate fi înregistrată ca urma secvențelor sale de acțiuni în care a fost implicat. Sunt definite multe operații utile cu urme. Un proces poate fi specificat înaintea implementării prin descrierea proprietăților urmelor sale. Sunt date reguli ce ajută la implementarea proceselor ce se pot dovedi corespunzătoare specificațiilor.

Al doilea capitol prezintă cum procesele pot fi asamblate în sisteme, în care componentele interacționează unele cu altele sau cu mediul extern. Introducerea concurenței nu introduce automat un factor de nedeterminism. Principalul exemplu din acest capitol este celebra problemă a celor cinci filozofi. A doua parte a cap. 2 prezintă adaptarea proceselor la situații noi prin schimbarea numelui evenimentelor în care sunt antrenate. Capitolul se încheie cu expunerea teoriei matematice a proceselor deterministe, incluzând elemente de teoria recursivității.

Al treilea capitol furnizează una din cele mai simple și cunoscute soluții pentru nedeterminism. Nedeterminismul este prezentat ca o tehnică de pătrundere a necunoscutului, din moment ce el rezultă natural din decizia de a ignora sau trece sub tăcere acele aspecte ale comportării sistemului de care nu suntem interesați. De asemenea se păstrează anumite simetrii cu determinismul în definiția operatorilor din teoria matematică. Demonstrațiile pentru procesele nedeterministe sunt ușor mai complicate decât cele de la procese deterministe, deoarece trebuie arătat că orice alegere posibil nedeterministă va avea o comportare corespunzătoare specificațiilor date. Din fericire, există tehnici pentru evitarea nedeterminismului și ele sunt folosite exclusiv în cap. 4 și 5. Astfel, studiul cap. 3 poate fi amânat până înainte de cap. 6, când introducerea nedeterminismului nu mai poate fi amânată.

În ultimele paragrafe ale cap. 3 se prezintă complet definirea matematică a conceptului de proces nedeterminist. Această parte va fi de interes pentru matematicianul care intenționează să exploreze fundamentelor subiectului sau

să verifice validitatea demonstrațiilor legilor algebrice și a celorlalte proprietăți ale proceselor. Programatorii pot privi legile ca ceva evident sau justificat de utilitatea lor, omițând fără probleme porțiunile mai teoretice.

Cap. 4 introduce în sfârșit comunicarea : un caz special de interacțiune a două procese, când unul emite un mesaj pe care celălalt, în același timp, îl recepționează. Avem astfel de a face cu o comunicare sincronizată. Dacă este necesară bufferarea pe un canal, aceasta se realizează prin interpunerea unui proces buffer între cele două procese. Un obiectiv major în proiectarea sistemelor concurente este viteza de răspuns în soluționarea problemelor practice. Aceasta este ilustrat prin modelarea unor algoritmi sistolici sau iterativi. Un caz simplu de comunicație este o conductă, definită ca o secvență de procese în care fiecare primește informații de la predecesor și transmite informații la succesor. Conducele sunt utile pentru implementarea unor protocoale de comunicație unidirecționale, structurate ca o ierarhie de nivele. În fine, importantul concept de tip abstract de dată este modelat cu un proces subordonat, fiecare instanță a lui comunicând numai cu blocul în care este declarat.

În cap. 5 se prezintă operatorii convenționali din programarea secvențială și încadrarea lor în modelul proceselor secvențiale comunicante. Poate apărea surprinzător pentru programatorii experimentați că acești operatori se bucură de același fel de elegante proprietăți algebrice ca și operatorii teoriilor matematice familiare, putându-se dovedi că programele secvențiale respectă specificațiile aproape la fel ca programele concurente, fiind definită chiar și o întrerupere externă pentru a fi utilizabilă în legile folosite.

Cap. 6 descrie cum se structurează și implementează un sistem în care un număr limitat de resurse fizice, ca de exemplu discurile sau imprimantele, pot fi partajate de mai multe procese, ale căror necesități de resurse variază în timp. Fiecare resursă este reprezentată ca un singur proces. De fiecare dată când o resursă este cerută de un proces utilizator, o nouă resursă virtuală este creată. O resursă virtuală se comportă ca și când ar fi subordonată procesului utilizator, comunicând cu resursa reală ori de câte ori e necesar. Aceste comunicații sunt înțelese cu acelea ale altor procese concurente active virtuale. Astfel procesele reale și virtuale joacă același rol ca și monitoarele și plicurile din Pascal Plus. Capitolul este ilustrat de o dezvoltare modulară și gradată a unor sisteme de operare atât simple cât și complete, cele mai mari exemple de altfel din carte.

Cap. 7 descrie un număr de alternative la concurență și comunicație care există deja și explică din punct de vedere practic, istoric și personal motivele care au dus la teoria expusă în capitolele precedente. Aici se recunoaște influența altor autori (lucrări), se dau recomandări și o introducere la lecturi următoare în domeniu.

Aprecieri

Este o mare plăcere să recunosc munca profundă și originală a lui Robin Milner, expusă în lucrarea sa clasică *Calculul proceselor comunicante*. Ideile sale originale, prietenia sa și rivalitatea profesională au fost surse constante de inspirație și încurajare pentru eforturile ce au culminat cu publicarea acestei cărți.

În ultimii douăzeci de ani m-au preocupat problemele legate de programarea concurentă și proiectarea limbajelor de programare. În această perioadă am beneficiat de colaborarea cu mari savanți ca Per Brinch Hansen, Stephen Brookes, Dave Bustard, Zhou Chao Chen, Ole-Johan Dahl, Edsger W. Dijkstra, John Elder, Jeremy Jacob, Ian Hayes, Jim Kaubisch, John Kennaway, T. Y. Kong, Peter Lauer, Mike McKeag, Carroll Morgan, Ernst-Rudiger Olderog, Rudi Reinecke, Bill Roscoe, Alex Teruel, Alastair Tocher și Jim Welsh.

În final, mulțumesc în mod deosebit lui O.-J. Dahl, E. W. Dijkstra, Leslie M. Goldschlager, Jeff Sanders și altora care cu migală au studiat acest text și-au scos la iveală erori sau obscurități. De asemenea mulțumesc participanților la școala de vară din Wollongong, secția programare, ianuarie 1983, studenților din Graduate School of Chinese Academy of Science, aprilie 1983 și studenților postgraduate în informatică din Oxford University, anii 1979-1984.

Glosar de simboluri

Logică

<i>Notatii</i>	<i>Semnificație</i>	<i>Exemple</i>
$=$	egal	$x=x$
\neq	diferit de	$x \neq x+1$
\square	sfârșitul unui exemplu sau demonstrație	
$P \wedge Q$	P și Q (conjuncție)	$x \leq x+1 \wedge x \neq x+1$
$P \vee Q$	P sau Q (disjuncție)	$x \leq y \vee y \leq x$
$\neg P$	P negat (negație)	$\neg 3 \geq 5$
$P \Rightarrow Q$	dacă P atunci Q	$x < y \Rightarrow x \leq y$
$P \Leftrightarrow Q$	P dacă și numai dacă Q	$x < y \Leftrightarrow y > x$
$\exists x.P$	există un x cu proprietatea P	$\exists x.x > y$
$\forall x.P$	pentru toți x cu proprietatea P	$\forall x.x < x+1$
$\exists x.A.P$	există un x din mulțimea A cu proprietatea P	
$\forall x.A.P$	pentru toți x din A cu proprietatea P	

Mulțimi

<i>Notatii</i>	<i>Semnificație</i>	<i>Exemple</i>
\in	aparține	Napoleon \in umanității
\notin	nu aparține	Napoleon \notin rușilor
$\{\}$	mulțime vidă	$\neg \text{Napoleon} \in \{\}$
$\{a\}$	mulțime cu un singur element (singleton)	$x \in \{a\} \Leftrightarrow x=a$
$\{a,b,c\}$	mulțime cu trei elemente	$c \in \{a,b,c\}$

$\{x P(x)\}$	mulțimea tuturor x cu proprietatea P	$\{a\} = \{x x=a\}$
$A \cup B$	A reunit cu B	$A \cup B = \{x x \in A \vee x \in B\}$
$A \cap B$	A intersectat cu B	$A \cap B = \{x x \in A \wedge x \in B\}$
$A - B$	A minus B	$A - B = \{x x \in A \wedge \neg x \in B\}$
$A \subseteq B$	A este inclus în B	$A \subseteq B = \forall x. A.x \in B$
$A \supseteq B$	A include B	$A \supseteq B = B \subseteq A$
$\{x:A P(x)\}$	mulțimea x din A cu $P(x)$	$\{0,1,2,\dots\}$
\mathbb{N}	mulțimea numerelor naturale	$PA = \{X X \subseteq A\}$
PA	mulțimea părților lui A	
$\bigcup_{n \geq 0} A_n$	reuniunea unei familii de mulțimi	$\bigcup_{n \geq 0} A_n = \{x \exists n \geq 0. x \in A_n\}$
$\bigcap_{n \geq 0} A_n$	intersecția unei familii de mulțimi	$\bigcap_{n \geq 0} A_n = \{x \forall n \geq 0. x \in A_n\}$

Funcții

Notafii	Semnificație	Exemple
$f:A \rightarrow B$	f este definită pe A cu valori în B	pătrat: $\mathbb{N} \rightarrow \mathbb{N}$
$f(x)$	un element din B ce corespunde lui x din A	
injecție	o funcție ce face să-i corespundă fiecărui element din A un element distinct din B	$x \neq y \Rightarrow f(x) \neq f(y)$
f^{-1}	inversa unei injecții f	$x = f(y) \Rightarrow y = f^{-1}(x)$
$\{f(x) P(x)\}$	mulțimea tuturor imaginilor lui x cu $P(x)$	$\{y \exists x. y = f(x), P(x)\}$
$f(C)$	imaginea lui C prin f	pătrat($\{3,5\}$) = $\{9,25\}$
$f \circ g$	f compus cu g	$f \circ g(x) = f(g(x))$
$\lambda x. f(x)$	funcția care face să-i corespundă fiecărei valori a lui x un $f(x)$	$(\lambda x. f(x))(3) = f(3)$

Urme

Paragraf	Notafii	Semnificație	Exemple
1.5	\diamond	urmă vidă	
1.5	$\langle a \rangle$	urmă cu element a	singleton
1.5	$\langle a, b, c \rangle$	urmă cu trei elemente	a apoi b apoi c
1.6.1	\wedge	concatenarea	$\langle a, b, c \rangle = \langle a, b \rangle \wedge \langle c \rangle$

1.6.1	s^n	s repetată de n ori	$\langle a, b \rangle^2 = \langle a, b, a, b \rangle$
1.6.2	$s \upharpoonright A$	s restricționată la A	$\langle b, c, d, a \rangle \upharpoonright \{a, c\} = \langle c, a \rangle$
1.6.5	$s \leq t$	s este un prefix al lui t	$\langle a, b \rangle \leq \langle a, b, c \rangle$
4.2.2	$s \leq^n t$	s coincide cu t mai puțin n simboluri	$\langle a, b \rangle \leq^2 \langle a, b, d, c \rangle$
1.6.5	s în t	s este conținut în t	$\langle c, d \rangle$ în $\langle b, c, d, a, b \rangle$
1.6.6	$\#s$	lungimea lui s	$\# \langle b, c, b, a \rangle = 4$
1.6.6	$s \downarrow b$	numărul de simboluri b în s	$\langle b, c, b, a \rangle \downarrow b = 2$
1.9.6	$s \downarrow c$	comunicația pe canalul c înregistrată în s	$\langle c.1, a.4, c.3, d.1 \rangle \downarrow c = \langle 1, 3 \rangle$
1.9.2	$\vee s$	s reunit	$\vee \langle \langle a, b \rangle, \langle c \rangle \rangle = \langle a, b, c \rangle$
1.9.7	s, t	s terminată cu succes și urmată de t	$(s \wedge \vee); t = s \wedge t$
1.6.4	A^*	mulțimi de secvențe cu elemente din A	$A^* = \{s \mid s \upharpoonright A = s\}$
1.6.3	s_0	capul secvenței s	$\langle a, b, c \rangle_0 = \langle a \rangle$
1.6.3	s'	coada secvenței s	$\langle a, b, c \rangle' = \langle b, c \rangle$
1.9.4	$s[i]$	elementul i din s	$\langle a, b, c \rangle[1] = b$
1.9.1	$f^*(s)$	f stea de s	pătrat [*] ($\langle 1, 5, 3 \rangle$) $= \langle 1, 25, 9 \rangle$
1.9.4	\overline{s}	inversa lui s	$\overline{\langle a, b, c \rangle} = \langle c, b, a \rangle$

Evenimente speciale

Paragraf	Notății	Semnificație
1.9.7	\vee	succes (terminare cu succes)
2.6.2	$l.a$	participarea la evenimentul a a procesului etichetat l
4.1	$c.v$	comunicarea valorii v pe canalul c
4.5	$l.c$	canalul c a procesului etichetat l
4.5	$l.c.v$	comunicarea mesajului v pe canalul $l.c$
5.4.1	\perp	catastrofă
5.4.3	\otimes	interschimbarea
5.4.4	\odot	punct de control pentru reluări
6.2	$ocupă$	acapararea unei resurse
6.2	$eliberează$	eliberarea unei resurse

675.673



Procese

Paragraf	Notatii	Semnificație
1.1	αP	alfabetul procesului P
4.1	ac	mulțimea mesajelor comunicabile pe canalul c
1.1.1	$a \rightarrow P$	a apoi P
1.1.3	$(a \rightarrow P b \rightarrow Q)$	a apoi P sau (alternativă) b apoi Q (dacă $a \neq b$)
1.1.3	$(x:A \rightarrow P(x))$	(alegerea lui x) din A apoi $P(x)$
1.1.2	$\mu X:A.F(X)$	procesul X cu alfabetul A astfel că $X=F(X)$
1.8.3	P/s	P după (angajarea în evenimentele urmei) s
2.3	$P Q$	P în paralel cu Q
2.6.2	$l:P$	P cu eticheta l
2.6.4	$L:P$	P cu etichete din mulțimea L
3.2	$P \sqcap Q$	P sau Q (nedeterminist)
3.3	$P \sqcup Q$	P sau Q
3.5	$P \backslash C$	P fără C (mascare)
3.6	$P Q$	P întretesut cu Q
4.4	$P \gg Q$	P înlănțuit cu Q
4.5	$P // Q$	P subordonat lui Q
6.4	$l::p/Q$	subordonare la distanță
5.1	$P;Q$	P (terminat cu succes) urmat de Q
5.4	$P \wedge Q$	P întrerupt de Q
5.4.1	$P \dashv Q$	P sau în caz de catastrofă Q
5.4.2	\hat{P}	P restartabil
5.4.3	$P \otimes Q$	P în alternanță cu Q
5.5	$P \triangleleft b \triangleright Q$	P dacă b altfel Q
5.1	$*P$	P repetat
5.5	$b * P$	cât timp b repetă P
5.5	$x := e$	x devine (valoarea) e
4.2	$b!e$	pe (canalul) b se emite (valoarea) e
4.2	$b?x$	din (canalul) b se preia x
6.2	$!!e?x$	apelul unei subrutine partajate cu numele l valoarea parametrului e și rezultatul în x
1.10.1	$P \text{ sat } S$	(procesul) P satisface (specificația) S
1.10.1	ur	o urmă arbitrară a procesului specificat
3.7	ref	un refuz arbitrar al procesului specificat
5.5.2	x	valoarea finală a lui x produsă de procesul specificat
5.5.1	$var(P)$	mulțimea variabilelor asignabile în P
5.5.1	$acc(P)$	mulțimea variabilelor accesibile în P

2.8.2	$P \sqsubseteq Q$	(determinist) Q poate face cel puțin cât și P
3.9	$P \sqsubseteq Q$	(nedeterminist) Q este cel puțin la fel de bun ca P sau mai bun
5.5.1	$\mathcal{D}e$	expresia e este definită

Algebră

Termen	Semnificație
reflexiv	o relație R astfel că xRx
antisimetric	o relație R astfel că $xRy \wedge yRx \Rightarrow x=y$
tranzitivă	o relație R astfel că $xRy \wedge yRz \Rightarrow xRz$
ordine parțială	o relație \leq care este reflexivă, antisimetrică și tranzitivă
limită	un cel mai mic element astfel că $\perp \leq x$
monotonă	o funcție f care respectă o ordine parțială $x \leq y \Rightarrow f(x) \Rightarrow f(y)$
strictă	o funcție identitate asupra limitei $f(\perp) = \perp$
idempotent	un operator binar f astfel că $xfx = x$
simetric	un operator binar f astfel că $xfy = yfx$
asociativ	un operator binar f astfel că $xf(yfz) = (xfy)fz$
distributivitate	f se distribuie cu g dacă $xf(ygz) = (xfy)g(xfz) \wedge (ygz)fx = (yfx)g(zfx)$
unitate	lui f este un element 1 astfel că $x/1 = 1fx = x$
zero	lui f este un element 0 astfel că $x/0 = 0fx = 0$

Grafuri

Termen	Semnificație
graf	o relație reprezentată printr-un desen
nod	un cerculeț în graf reprezentând un element din domeniul de definiție sau de valori
arc	o linie simplă sau cu o săgeată într-un graf ce conectează două noduri între care are loc relația
graf neorientat	graf cu o relație simetrică
graf orientat	graf al unei relații nesimetrice reprezentat prin arce cu săgeți
ciclu orientat	o mulțime de noduri legate prin arce cu săgeți orientate în același sens

ciclu neorientat

**o mulțime de noduri legate prin arce simple sau
cu săgeți în sensuri contrare**

Procese

1.1 Introducere

Să uităm puțin modul cum privim calculatoarele, programarea și să ne gândim la obiectele din lumea înconjurătoare, care interacționează între ele și cu noi sau au diferite reguli de comportare. Să ne gândim la ceasuri, numărătoare, telefoane, jocuri electronice, automate de vânzare. Pentru a descrie regulile lor de comportare trebuie mai întâi să decidem ce fel de evenimente sau acțiuni ne interesează. Totodată să alegem un nume diferit pentru fiecare tip. În cazul unui automat de vânzare simplu există două feluri de evenimente :

mon introducerea unei monezi în automat

choc extragerea unei ciocolate din aparat

În cazul unui automat mai complex pot fi deosebite o varietate de evenimente:

in1p introducerea unei monezi de un penny

in2p introducerea unei monezi de doi penny

mic extragerea unui biscuit sau napolitane mici

mare extragerea unui biscuit sau napolitane mari

ex1p extragere restului de un penny

Înțelegem că fiecare nume de eveniment cuprinde o *clasă* de evenimente, adică aparițiile lui în timp. Distincția între clasă și apariție este analoagă, de exemplu, în cazul literei c, ce apare, evident, de foarte multe ori în această carte.

Mulțimea numelor evenimentelor ce sunt considerate relevante pentru o anumită descriere a unui obiect se numește *alfabet*. Alfabetul este o proprietate apriori și permanentă a unui obiect. Din punct de vedere logic este imposibil pentru un obiect să se angajeze într-un eveniment ce nu face parte din alfabetul său. De exemplu, un automat de vândut ciocolată nu va putea dintr-o dată să vândă jucării. S-ar putea ca automatul să nu furnizeze ciocolată

deoarece este defect, sau ciocolata s-a terminat, sau nimeni nu dorește să o cumpere. Atât timp însă cât alfabetul său conține evenimentul *choc*, alfabetul rămâne neschimbat chiar dacă evenimentul nici măcar nu apare.

Alegerea unui alfabet presupune o etapă deliberată de simplificare, de ignorare a proprietăților și acțiunilor considerate de un interes mai scăzut: De exemplu, culoarea, greutatea, forma automatului nu se consideră relevante, de asemenea alimentarea cu ciocolată sau golirea casetei de monezi, sunt considerate fără interes pentru cumpărător.

Apariția fiecărui eveniment în existența unui obiect se consideră o acțiune instantanee, atomică, fără durată. Acțiuni extinse sau consumatoare de timp pot fi reprezentate ca o pereche de evenimente, primul corespunzător începutului duratei de timp iar al doilea corespunzător sfârșitului ei. Durata unei acțiuni este dată de intervalul dintre aparițiile evenimentelor de început și de sfârșit. În timpul unui astfel de interval pot apărea diverse alte evenimente. Două acțiuni extinse se pot suprapune în timp dacă începutul uneia precede sfârșitul celeilalte.

Alt detaliu ignorat deliberat este momentul apariției evenimentelor. Avantajul rezidă în simplificarea proiectării și a raționamentelor, ele putându-se aplica pentru sisteme software în orice context de viteză sau performanțe. În cazul în care momentele aparițiilor sunt critice, ele pot fi tratate independent de corectitudinea logică a modelului. De altfel, independența de timp a fost o condiție permanent necesară pentru succesul limbajelor de programare de nivel înalt.

O consecință a ignorării timpului este refuzul considerării simultaneității evenimentelor. Când devine totuși esențială simultaneitatea unei perechi de evenimente, (ca de ex. în sincronizare) perechea se reprezintă printr-un singur eveniment, în celelalte cazuri perechea reprezentându-se prin două evenimente, ce se succed într-o ordine dată.

La constituirea alfabetului nu se va face distincție între evenimentele ce țin de obiect (de ex. *choc*) și cele care țin de un agent exterior obiectului (de ex. *mon*). Evitarea cauzalității conduce astfel la o semnificativă simplificare a teoriei și a exemplificărilor ei.

Din acest moment vom folosi noțiunea de *proces* pentru a defini comportarea unui obiect având în vedere evenimentele ce compun alfabetul său. Vom folosi următoarele convenții :

1. Evenimentele vor fi reprezentate prin cuvinte scrise cu litere mici, de exemplu : *mon*, *choc*, *in2p*, *ex1p*, sau pur și simplu prin litere mici, de exemplu : *a*, *b*, *c*, *d*, *e*.
2. Procesele vor fi reprezentate prin cuvinte scrise cu majuscule, de exemplu : *AVS* automat de vândut simplu *AVC* automat de vândut complex sau prin majuscule simple, de exemplu : *P*, *Q*, *R*, ce desemnează în legi procese arbitrare.

3. Literele x, y, z sunt variabile cu semnificație de evenimente.
4. Literele A, B, C corespund unor mulțimi de evenimente (alfabete).
5. Literele X, Y sunt variabile ce semnifică procese.
6. Alfabetul unui proces este notat αP , de exemplu :

$$\alpha AVS = \{mon, choc\}$$

$$\alpha AVC = \{in1p, in2p, mic, mare, ex1p\}$$

Procesul cu alfabetul A care nu se angajează în nici un eveniment din alfabetul său se numește $STOP_A$. Acest proces descrie comportarea unui obiect care și-a terminat activitatea. Cu toate că există evenimente în alfabetul său, deci ar exista posibilitatea angajării în unele dintre ele, acest lucru nu se mai întâmplă. De asemenea, obiecte cu alfabete diferite devin diferite, independent de angajarea în evenimente. Astfel $STOP_{\alpha AVS}$ este diferit de $STOP_{\alpha AVC}$, deoarece alfabetele lor nu sunt identice.

Vom trece la definirea unor notații simple pentru a putea descrie obiecte ce se angajează în diverse evenimente.

1.1.1 Prefix

Fie x un eveniment și P un proces. Atunci

$$(x \rightarrow P) \quad (\text{semnifică "x apoi P"})$$

descrie un obiect care întâi se angajează în evenimentul x și apoi se comportă ca și P . Procesul $STOP_{\alpha AVS}$ este definit cu același alfabet ca și P , de aceea x trebuie să aparțină neapărat alfabetului (αP), sau formal

$$\alpha(x \rightarrow P) = \alpha P \quad \text{dacă } x \in \alpha P$$

Exemple

X1 Un automat de vânzare simplu care primește o monedă și apoi se strică ("o înghite")

$$(mon \rightarrow STOP_{\alpha AVS}) \quad \square$$

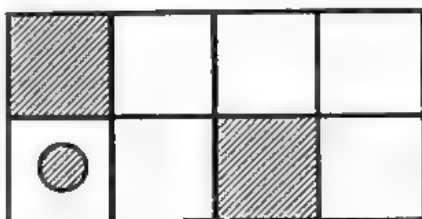
X2 Un automat simplu care este folosit cu succes de doi clienți înainte de a se strica

$$(mon \rightarrow (choc \rightarrow (mon \rightarrow (choc \rightarrow STOP_{\alpha AVS}))))$$

După cum se observă, *AVS* va accepta o monedă după care va furniza o ciocolată, iar va accepta o monedă și iar va furniza o ciocolată, în final oprindu-se. Automatul nu va putea accepta două monezi consecutiv, furnizând două ciocolate (rezultă din funcționarea automatului care blochează mecanic introducerea celei de-a doua monezi până la eliberarea ciocolatei pentru moneda anterioară). \square

Pe viitor se va renunța la paranteze în cazul unei succesiuni de evenimente ca acelea din *X2*, cu convenția că \rightarrow este asociativ la dreapta.

X3 Un marker, pornind din colțul stânga jos al unei suprafețe riglate, se poate mișca numai la dreapta sau în sus într-o căsuță adiacentă



$$\alpha MSR = \{sus, dreapta\}$$

$$MSR = (dreapta \rightarrow sus \rightarrow dreapta \rightarrow dreapta \rightarrow STOP_{\alpha MSR})$$

 \square

Trebuie observat că operatorul \rightarrow presupune un proces în dreapta și *un singur eveniment* în stânga. Astfel, dacă *P* și *Q* sunt procese este incorect sintactic

$$P \rightarrow Q$$

Metoda prin care se descrie un proces care inițial se comportă ca *P* și apoi ca *Q* se va prezenta în cap. 4. De asemenea, dacă *x* și *y* sunt evenimente, este incorect sintactic

$$x \rightarrow y$$

Putem avea însă pentru un proces

$$x \rightarrow (y \rightarrow STOP)$$

Trebuie să distingem cu atenție conceptul de eveniment de acela de proces care se angajează în evenimente – pot fi mai multe sau nici unul.

1.1.2 Recursivitatea

Notăția prefix poate fi folosită pentru a descrie comportarea completă a unui proces care eventual se și oprește. Dar devine anevoios de a descrie comportarea unui automat care funcționează ciclic. De aceea avem nevoie să descriem tipuri de comportări repetitive printr-o notație mult mai scurtă. Aceste notații nu ar trebui să țină cont de durata comportării ciclice a obiectului. Considerăm un ceas care ticăie

$$\alpha CEAS = \{tic\}$$

Să considerăm un obiect ceas care se comportă exact ca ceasul inițial, dar emite mai întâi un *tic*

$$(tic \rightarrow CEAS)$$

Evident comportarea acestui obiect nu poate fi deosebită de cea a ceasului original. Acest raționament ne conduce la ecuația

$$CEAS = (tic \rightarrow CEAS)$$

Această ecuație poate fi privită ca definind comportarea unui ceas, în același mod în care rădăcina pătrată din doi poate fi privită ca fiind soluția pozitivă a ecuației

$$x = x^2 + x - 2$$

Se pot scrie ecuații obținute din cea inițială prin substituirea în dreapta a membrului stâng

$$\begin{aligned} CEAS &= (tic \rightarrow CEAS) && \text{ecuație inițială} \\ &= (tic \rightarrow (tic \rightarrow CEAS)) && \text{substituție} \\ CEAS &= (tic \rightarrow tic \rightarrow tic \rightarrow CEAS) && \text{similar} \end{aligned}$$

După cum se observă, membrul drept poate fi desfășurat cât vrem, având de-a face cu o comportare nemărginită

$$tic \rightarrow tic \rightarrow tic \rightarrow \dots$$

la fel cum rădăcina pătrată din doi poate fi văzută ca limită a numărului de zecimale

$$1,414\dots$$

Descrierea recursivă de mai sus va fi efectivă numai dacă membrul drept începe cu cel puțin un eveniment prefixat tuturor aparițiilor recursive ale numelui procesului. De exemplu, ecuația recursivă

$$X=X$$

nu este efectivă din punctul nostru de vedere, deoarece orice proces poate fi o soluție a sa. O descriere a unui proces care începe cu un prefix se spune că este cu *gardă*. Dacă $F(X)$ este o expresie cu gardă conținând numele procesului X , iar A este alfabetul său, ecuația

$$X=F(X)$$

are soluție unică dându-se alfabetul A . Vom nota soluția prin expresia

$$\mu X:A.F(X)$$

X poate fi privit ca o variabilă locală și prin urmare poate fi schimbat

$$\mu X:A.F(X) = \mu Y:A.F(Y)$$

Această egalitate este justificată de faptul că o soluție pentru X din ecuația

$$X=F(X)$$

este și o soluție a ecuației

$$Y=F(Y)$$

Pe viitor, vom da definițiile recursive ale proceselor atât cu ajutorul ecuațiilor, cât și cu ajutorul lui μ , evident mai convenabil. În cazul că folosim varianta cu $\mu X:A.F(X)$, vom omite menționarea explicită a alfabetului A , dacă acesta este evident din contextul sau conținutul procesului.

Exemple

X1 Funcționarea unui ceas va fi descrisă de

$$CEAS = \mu X: \{t|c\}.(t|c \rightarrow X)$$

□

X2 Un automat de vândut ciocolată care funcționează non-stop

$$AIVS = (mon \rightarrow (choc \rightarrow AIVS))$$

După cum s-a menționat mai sus, această ecuație este o altă formă de scriere pentru definirea cea mai formală

$$AIVS = \mu X. \{mon, choc\}. (mon \rightarrow (choc \rightarrow X)) \quad \square$$

X3 Un automat care schimbă o monedă de cinci penny în monezi de un penny și doi penny

$$\begin{aligned} \alpha SCH5A &= \{in5p, ex1p, ex2p\} \\ SCH5A &= (in5p \rightarrow ex2p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5A) \end{aligned} \quad \square$$

X4 Tot un automat de schimbat monezi, dar cu altă succesiune

$$SCH5B = (in5p \rightarrow ex1p \rightarrow ex1p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5B) \quad \square$$

Faptul că ecuațiile cu gardă au o soluție și ea este unică poate fi arătat prin metoda substituției. De fiecare dată când membrul drept al ecuației este substituit numelui procesului, se extinde definirea comportării oricât de mult, astfel că orice comportare finită poate fi determinată în acest mod. Putem extrapola prin urmare, că două procese care au aceeași comportare până la un moment oarecare finit de timp, sunt de fapt același proces. O demonstrație formală, completă, se va face odată cu definirea matematică a proceselor, dată în 2.8.3. Se va da și o semnificație pentru recursivitatea fără gardă în 3.8.

1.1.3 Alegere

După cum s-a văzut, cu ajutorul recursivității și a prefixului este posibil de a descrie obiecte cu un singur fir de comportare, fără ramificații. Totuși, multe obiecte interacționează cu mediul, ceea ce se reflectă asupra comportării lor prin aceea că trebuie făcută o alegere la un moment dat. De exemplu, un automat de vânzare poate oferi două canale de introdus monezi, unul pentru un penny, celălalt pentru doi penny. Cumpărătorul trebuie să decidă angajarea în unul din cele două evenimente. Dacă x și y sunt două evenimente distincte

$$(x \rightarrow P | y \rightarrow Q)$$

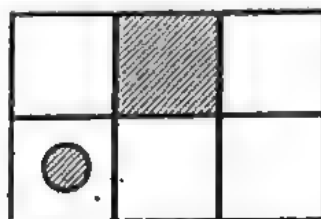
descrie un obiect care se angajează fie în evenimentul x fie în y . Comportarea ulterioară a obiectului este descrisă de P , dacă evenimentul care a avut loc a fost x , sau de Q , dacă evenimentul care a avut loc a fost y . Deoarece x și y sunt evenimente diferite, alegerea între P și Q este determinată de primul care a avut loc. Referitor la alfabetul avem

$$\alpha(x \rightarrow P | y \rightarrow Q) = \alpha P \quad \text{dacă } \{x, y\} \subseteq \alpha P \text{ și } \alpha P = \alpha Q$$

Semnul $|$ va semnifica "alegere, alternativă cu, sau" : "alegere între x apoi P și y apoi Q ".

Exemple

X1 Posibilele mișcări ale markerului pe suprafața riglată sunt definite de procesul



$$(sus \rightarrow STOP | dreapta \rightarrow dreapta \rightarrow sus \rightarrow STOP)$$

□

X2 Un automat de schimbat monezi cu două posibilități de schimb pentru cinci penny (a se compara cu 1.1.2 X3 și X4, unde nu aveam posibilități de alegere)

$$SCH5C = in5p \rightarrow (ex1p \rightarrow ex1p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5C) \\ | ex2p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5C)$$

□

X3 Un automat care poate da ciocolată sau bomboane

$$AVCB = \mu X. mon \rightarrow (choc \rightarrow X | bonbon \rightarrow X)$$

□

X4 Un automat mai complex, care permite introducerea mai multor feluri de monezi și furnizează atât napolitane mari și mici cât și restul aferent

$$AVC = (in2p \rightarrow (mare \rightarrow AVC \\ | mic \rightarrow ex1p \rightarrow AVC) \\ | in1p \rightarrow (mic \rightarrow AVC \\ | in1p \rightarrow (mare \rightarrow AVC \\ | in1p \rightarrow STOP))))$$

Deoarece are o funcționare mai complicată, existând riscul de a introduce monezi și a nu obține nimic, automatul ar trebui însoțit de observații de forma

"ATENȚIE : Nu introduceți trei monezi de un penny la rând."

□

X5 Un automat care întâi dă ciocolata, având încredere că apoi va fi plătită. Secvența normală plată–ciocolată este de asemenea implementată

$$AI\ MCCM = \mu X. (mon \rightarrow choc \rightarrow X) \\ | choc \rightarrow mon \rightarrow X) \quad \square$$

X6 Pentru a preveni pierderile putem impune automatului o plată inițială, profitând de $AI\ MCCM$

$$AI\ S2 = (mon \rightarrow AI\ MCCM)$$

Acest automat va permite introducerea a până la două monezi consecutive înainte de a furniza două ciocolate una după alta. În schimb, nu va elibera mai multe ciocolate decât monezi introduse. \square

X7 Un proces de copiere se angajează în următoarele evenimente

pri.0 primește un zero pe canalul său de intrare
pri.1 primește un unu pe canalul său de intrare
emi.0 emite un zero pe canalul său de ieșire
emi.1 emite un unu pe canalul său de ieșire

Comportarea sa constă din repetarea unor perechi de evenimente. În fiecare ciclu, primește un bit pe care îl emite la ieșire

$$COPIEBIT = \mu X. (pri.0 \rightarrow emi.0 \rightarrow X \\ | pri.1 \rightarrow emi.1 \rightarrow X)$$

După cum se observă, procesul permite mediului de a face alegerea valorii care se primește, dar nu și a celei care se copie. Aceasta comportare se deosebește de tratarea comunicației din cap. 4. \square

Definirea alegerii se poate ușor extinde la mai mult de două posibilități

$$(x \rightarrow P | v \rightarrow Q | \dots | z \rightarrow R)$$

Trebuie remarcat că operatorul $|$ pentru alegere nu se aplică proceselor, fiind incorect sintactic o expresie de forma $P|Q$, unde P și Q sunt procese. Motivul acestei constrângeri este eliminarea expresiilor de forma

$$(x \rightarrow P) | (x \rightarrow Q)$$

care numai în aparență oferă o alegere. Această expresie își va găsi rostul odată cu introducerea nedeterminismului, în paragraful 3.3. Pentru contextul actual, dacă x, y, z sunt evenimente diferite

$$(x \rightarrow P | y \rightarrow Q | z \rightarrow R)$$

poate fi privit ca un singur operator cu trei argumente, P, Q, R . Menționăm ca incorect sintactic și o expresie de forma

$$(x \rightarrow P | (y \rightarrow Q | z \rightarrow R))$$

În general, dacă B este o mulțime de evenimente și $P(x)$ este o expresie definind un proces pentru fiecare x diferit din B , atunci

$$(x: B \rightarrow P(x))$$

definește un proces care întâi oferă șansa oricărui eveniment y din B și apoi se comportă ca P . Se poate spune " x din B apoi P de x ". În această construcție x este o variabilă locală, deci

$$(x: B \rightarrow P(x)) = (y: B \rightarrow P(y))$$

Vom spune că mulțimea B definește *meniul* inițial al procesului, deoarece include acțiuni între care trebuie făcută o alegere la începutul observării comportării.

Exemplu

X8 Un proces care se angajează oricând, în orice eveniment din alfabetul său notat A

$$\begin{aligned} \alpha RUN_A &= A \\ RUN_A &= (x: A \rightarrow RUN_A) \end{aligned}$$

□

În cazul special al mulțimii B formate numai din evenimentul e , avem

$$(x: \{e\} \rightarrow P(x)) = (e \rightarrow P(e))$$

deoarece e este singurul eveniment inițial posibil. În cazul extrem când meniul inițial este mulțime vidă, nu se poate întâmpla nimic

$$(x: \{\} \rightarrow P(x)) = (y: \{\} \rightarrow P(y)) = STOP$$

Operatorul de alegere poate fi acum redefinit, folosind meniul, astfel

$$(a \rightarrow P | b \rightarrow Q) = (x: B \rightarrow R(x))$$

unde $B = \{a, b\}$ și $R(x) = \text{if } x=a \text{ then } P \text{ else } Q$

O alegere între trei sau mai multe evenimente se poate exprima similar. Alegerea, prefixul și *STOP* sunt de fapt cazuri speciale ale alegerii pe bază de meniu. Această observație va fi de real folos în formularea legilor generale ale proceselor (paragraful 1.3), precum și la implementarea lor (paragraful 1.4).

1.1.4 Recursivitatea mutuală

Recursivitatea cum a fost introdusă permite definirea unui singur proces ca soluție a unei singure ecuații. Metoda poate fi generalizată la un sistem de mai multe ecuații cu mai multe necunoscute. Pentru aceasta, trebuie ca toți membrii dreptei ai ecuațiilor să fie cu gardă iar necunoscutele (nume de procese) trebuie să apară o singură dată în câte un membru stâng al unei ecuații.

Exemplu

X1 Un automat de băuturi are două butoane etichetate ANANAS și KIWI. Acțiunile de apăsare a butoanelor se notează cu *setananas* și *setkiwi*, iar acțiunile de eliberare a băuturilor cu *ananas* și *kiwi*. Alegerea unei băuturi se face numai prin apăsarea butonului corespunzător înainte de a o obține. Iată cum arată alfabetul și ecuațiile definind comportarea procesului *AK*. Sînt utilizate două procese auxiliare *A* și *K*, definite recursiv

$$\alpha AK = \alpha A = \{setananas, setkiwi, ananas, kiwi\}$$

$$AK = (setananas \rightarrow A | setkiwi \rightarrow K)$$

$$A = (ananas \rightarrow A | setkiwi \rightarrow K | setananas \rightarrow A)$$

$$K = (kiwi \rightarrow K | setananas \rightarrow A | setkiwi \rightarrow K)$$

După primul eveniment (apăsarea unui buton), automatul este în una din stările *A* sau *K*. În fiecare stare automatul poate servi băutura corespunzătoare sau poate fi comutat în cealaltă stare. Apăsarea butonului aceleiași stări este permisă și are efectul de a menține starea. \square

Pentru sisteme mari de ecuații este posibilă folosirea variabilelor indexate

Exemplu

X2 Un obiect pornește de pe sol și poate face o mișcare în *sus*. Ulterior poate efectua mișcări în *sus* sau *jos*, cu excepția cazului când se află pe sol și nu poate face decât mișcarea *sus*. Totuși când este pe sol, obiectul mai poate efectua și mișcarea *pe_loc*. Fie n un număr natural, deci din $\{0,1,2,\dots\}$. Notăm cu MM_n , (marker mobil), comportarea obiectului când se află la n mișcări depărtare de sol. Comportarea sa inițială este descrisă de

$$MM_0 = (sus \rightarrow MM_1 | pe_loc \rightarrow MM_0)$$

iar restul ecuațiilor, în număr infinit, sunt

$$MM_{n+1} = (sus \rightarrow MM_{n+2} | jos \rightarrow MM_n)$$

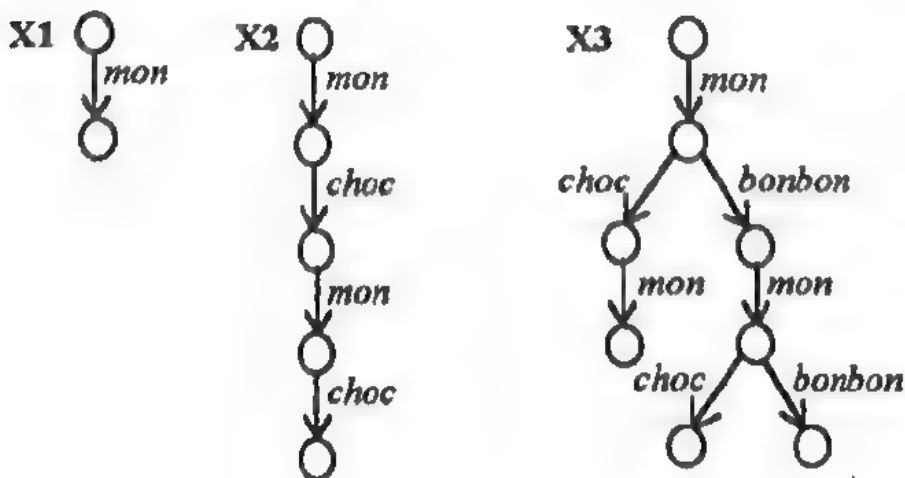
unde n este număr natural, după cum s-a specificat.

Cu toate că ecuația generală cuprinde pe MM_n, MM_{n+1}, MM_{n+2} , nu se poate folosi vreo metodă inductivă pentru a afla pe MM_n , deoarece el apare în ambii membri. Tot ceea ce se poate spune este că avem de-a face cu un sistem infinit de ecuații cu necunoscutele definite mutual recursiv, membrii dreپți ai ecuațiilor fiind cu gardă, deci valizi din punctul nostru de vedere. \square

1.2 Reprezentări grafice

Este util uneori de a reprezenta grafic un proces ca o structură arborescentă, formată din noduri și arce. În terminologia curentă a automatelor, nodurile reprezintă stări iar arcele tranziții. Există un nod rădăcină, starea inițială, iar fiecare arc va fi etichetat cu evenimentul care este angajat în respectiva tranziție. Arcele ce pornesc din același nod trebuie să aibă etichete diferite.

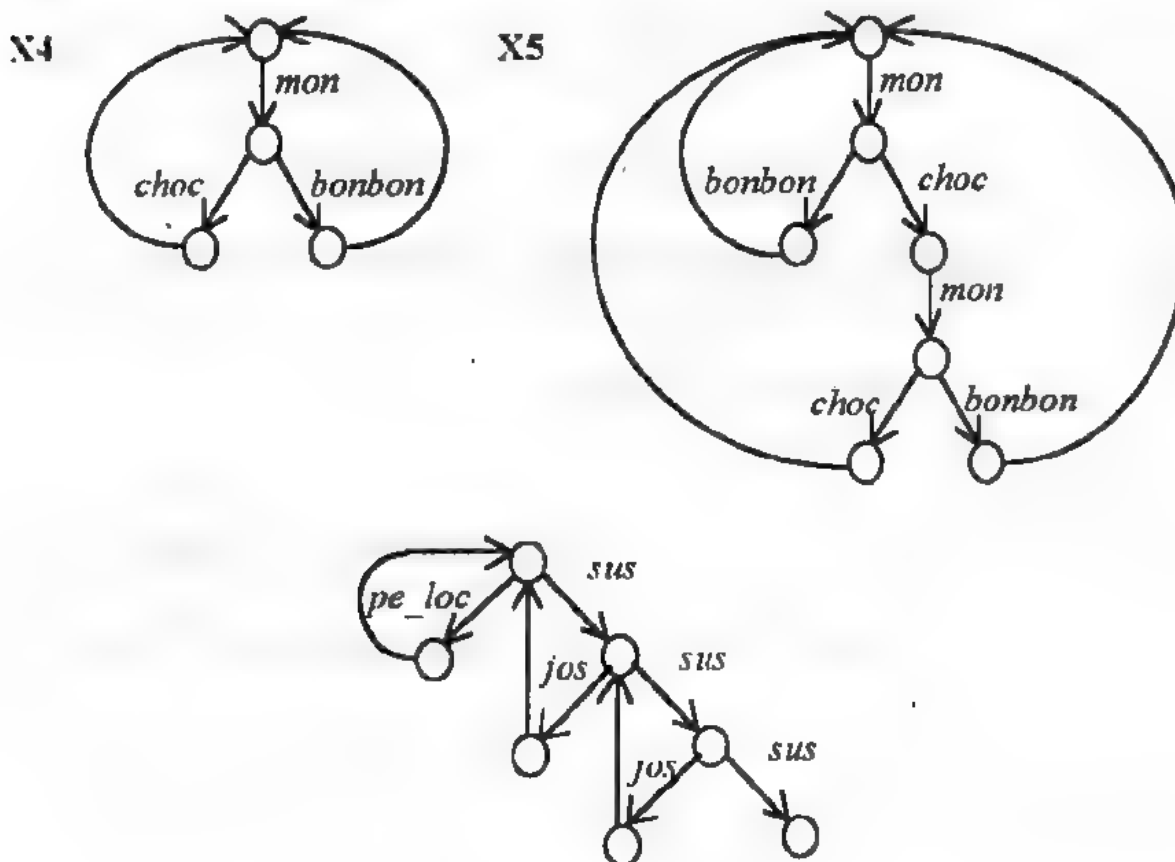
Exemple (1.1.1 X1, X2; 1.1.3 X3)



\square

În aceste trei exemple starea finală este *STOP*, reprezentată de nodurile frunză. Pentru reprezentarea proceselor cu o comportare infinită trebuie introdusă convenția existenței unui arc neetichetat de la nodurile frunză la noduri strămoși. Tranzițiile neetichetate se consideră a se produce imperceptibil și imediat, ajungându-se în nodurile din care pleacă respectivele arce. \square

De fapt, cele două reprezentări ilustrează același proces (1.1.3 X3), fiind clară slăbiciunea reprezentărilor grafice de a demonstra egalitatea proceselor. O altă slăbiciune a reprezentărilor grafice este în cazul proceselor cu un număr infinit de stări, de exemplu $\Lambda\Lambda I_0$.



1.3 Legi

Chiar în condițiile numărului mic de notații introduse până în prezent sunt destule posibilități de a descrie aceeași comportare. De exemplu, nu contează ordinea prezentării evenimentelor într-o alegere

$$(x \rightarrow P \mid y \rightarrow Q) = (y \rightarrow Q \mid x \rightarrow P)$$

De asemenea este evident că un proces care nu mai poate avea nici o comportare este diferit de unul ce se angajează în cel puțin un eveniment

$$(x \rightarrow P) \neq STOP$$

Pentru a înțelege corect notațiile și a opera cu ele, trebuie să învățăm a recunoaște care expresii descriu același obiect și care nu, la fel cum toți știm că expresia $(x+y)$ este tot una cu $(y+x)$. Identitatea sau nu a proceselor cu același alfabet poate fi dovedită prin apelarea la legi algebrice asemănătoare celor aritmetice.

Prima dintre ele (L1 de mai jos) se ocupă de operatorul alegere. Ea asigură că două procese descrise cu ajutorul alegerii sunt diferite dacă oferă șanse diferite în primul pas sau dacă după același prim pas, ele se comportă diferit. Dacă totuși alegerile inițiale sunt aceleași și dacă pentru fiecare alegere inițială comportările ulterioare sunt aceleași, atunci evident procesele sunt identice.

$$L1 \quad (x.A \rightarrow P(x)) = (y.B \rightarrow Q(y)) \iff (A=B \wedge \forall x \in A. P(x) = Q(y))$$

De acum încolo vom presupune fără a o face explicit că alfabetele proceselor din cei doi membri ai unei ecuații sunt identice

Legea L1 are un număr de consecințe

$$L1A \quad STOP \neq (d \rightarrow P)$$

$$\text{Demonstrație} \quad MS = (x: \{ \} \rightarrow P)$$

$$\neq (x: \{ d \} \rightarrow P)$$

$$= MD$$

prin definiție (sfârșitul lui 1.1.3)

deoarece $\{ \} \neq \{ d \}$

prin definiție (sfârșitul lui 1.1.3)

$$L1B \quad (c \rightarrow P) \neq (d \rightarrow Q)$$

dacă $c \neq d$

$$\text{Demonstrație} \quad \{ c \} \neq \{ d \}$$

$$L1C \quad (c \rightarrow P | d \rightarrow Q) = (d \rightarrow Q | c \rightarrow P)$$

$$\text{Demonstrație} \quad \text{Definim} \quad R(x) = P$$

$$= Q$$

dacă $x = c$

dacă $x = d$

$$MS = (x: \{ c, d \} \rightarrow R(x))$$

$$= (x: \{ d, c \} \rightarrow R(x))$$

$$= MD$$

prin definiție

deoarece $\{ c, d \} = \{ d, c \}$

prin definiție

$$L1D \quad (c \rightarrow P) = (c \rightarrow Q) = P = Q$$

$$\text{Demonstrație} \quad \{ c \} = \{ c \}$$

Aceste legi permit demonstrarea unor teoreme simple.

Exemple

X1 $(mon \rightarrow choc \rightarrow mon \rightarrow choc \rightarrow STOP) \neq (mon \rightarrow STOP)$

Demonstrație: din L1D și L1A

□

X2 $\mu X.(mon \rightarrow (choc \rightarrow X) \mid bonbon \rightarrow X) = \mu X.(mon \rightarrow (bonbon \rightarrow X) \mid choc \rightarrow X)$

Demonstrație:

din L1C

□

Pentru a putea demonstra teoreme mai generale privind recursivitatea proceselor este necesar de a introduce o lege care stabilește existența unei soluții unice în cazul ecuațiilor recursive cu gardă efectivă.

L2 Dacă $F(X)$ este o expresie cu gardă

$$(Y = F(Y)) \equiv (Y = \mu X.F(X))$$

O consecință imediată și importantă este că $\mu X.F(X)$ este de asemenea o soluție

$$\text{L2A } \mu X.F(X) = F(\mu X.F(X))$$

Exemplu

X3 Fie $AV1 = (mon \rightarrow AV2)$

și $AV2 = (choc \rightarrow AV1)$

Trebuie demonstrat că $AV1 = AVS$

Demonstrație $AV1 = (mon \rightarrow AV2)$

$$= (mon \rightarrow (choc \rightarrow AV1))$$

definiția lui $AV1$

definiția lui $AV2$

Deci $AV1$ este și ea o soluție a ecuației recursive descrisă de AVS . Deoarece ecuația este cu gardă efectivă are o singură soluție, $AV1$ și AVS fiind două nume diferite pentru aceeași soluție.

Această teoremă apare ca evident adevărată. Singurul scop al demonstrației este de a arăta printr-un exemplu că legile sunt suficient de puternice pentru a stabili fapte de acest gen. Când se demonstrează fapte evidente din legi mai puțin evidente este important de a justifica fiecare linie din demonstrație, ca o verificare că demonstrația nu folosește concluziile drept ipoteze.

□

Legea L2 poate fi extinsă la recursivitatea mutuală. Un sistem de ecuații recursive mutual poate fi scris în formă generală folosind indici

$$X_i = F(i, X) \quad \text{pentru toți } i \in S$$

unde S este mulțimea indecșilor ecuațiilor, cu un singur index per ecuație
 și X este un masiv de procese cu indici în domeniul mulțimii S
 și $F(i, X)$ este o expresie cu gardă efectivă.

În aceste condiții legea L3 precizează că există un singur masiv de procese X , ale cărui elemente satisfac toate ecuațiile

L3 În condițiile de mai sus

dacă $(\forall i: S. (X_i = F(i, X) \wedge Y_i = F(i, Y)))$ atunci $X = Y$

1.4 Implementarea proceselor

În condițiile notațiilor introduse până acum orice proces P poate fi scris sub forma

$$(x: B \rightarrow F(x))$$

unde F este o funcție definită pe mulțimea simbolurilor proceselor, cu valori în mulțimea proceselor și B poate fi vidă (în cazul lui *STOP*), poate conține un singur element (în cazul prefixului), sau poate conține mai multe elemente (în cazul alegerii). Pentru procesele definite recursiv am insistat că recursivitatea trebuie să fie cu gardă efectivă astfel că poate fi scrisă

$$\mu X. (x: B \rightarrow F(x, X))$$

iar desfășurat, ia următoarea formă folosind L2A

$$(x: B \rightarrow F(x, \mu X. (x: B \rightarrow F(x, X))))$$

Având în vedere cele arătate, un proces poate fi privit ca o funcție F , cu domeniul B , format dintr-o mulțime de evenimente în care procesul se poate angaja inițial. Pentru fiecare eveniment x din B , $F(x)$ definește comportarea ulterioară a procesului după angajarea în evenimentul x .

Aceste observații permit reprezentarea unui proces ca o funcție într-un limbaj de programare funcțional precum LISP-ul. Fiecare eveniment din alfabetul procesului se va reprezenta ca un atom, de exemplu "MON", "BONBON". Un proces devine o funcție cu argumente astfel de simboluri. Dacă simbolul *nu* este un prim eveniment posibil pentru proces, funcția returnează

un simbol special numit "BLIP", folosit numai în acest scop. De exemplu, deoarece *STOP* nu se angajează în nici un eveniment, poate fi definit

$$STOP = \lambda x. "BLIP"$$

Dacă însă argumentul este un eveniment posibil pentru proces se returnează ca rezultat altă funcție, și anume comportarea ulterioară a procesului. Astfel

$(mon \rightarrow STOP)$ se reprezintă ca o funcție

$$\lambda x. \text{ if } x = "MON" \text{ then } STOP \\ \text{ else } "BLIP"$$

Acest ultim exemplu ne arată facilitatea LISP-ului de a returna o funcție (*STOP*) ca rezultat al unei alte funcții. LISP-ul permite de asemenea transmiterea unei funcții ca argument, al unei alte funcții, facilitate folosită în reprezentarea operatorului prefix ($c \rightarrow P$)

$$prefix(c, P) = \lambda x. \text{ if } x = c \text{ then } P \\ \text{ else } "BLIP"$$

O funcție care să reprezinte alegerea generală binară ($c \rightarrow P \mid d \rightarrow Q$) necesită patru parametri

$$alegere2(c, P, d, Q) = \lambda x. \text{ if } x = c \text{ then } P \\ \text{ else if } x = d \text{ then } Q \\ \text{ else } "BLIP"$$

Procese definite recursiv pot fi reprezentate cu ajutorul facilității *LABEL* din LISP. De exemplu, automatul simplu de vânzare ($\mu X. mon \rightarrow choc \rightarrow X$) se reprezintă ca

$$LABEL\ X. prefix("MON, prefix("CHOC, X))$$

LABEL poate fi folosit pentru a implementa și recursivitatea mutuală. De exemplu, MM_n (1.1.4 X2) poate fi considerată ca o funcție pe mulțimea numerelor naturale și având codomeniu procese (la rândul lor funcții). Astfel MM_n poate fi definită

$$MM = LABEL\ X. \lambda n. \\ \text{ if } n = 0 \text{ then } alegere2("LATERAL, X(0), "SUS, X(1)) \\ \text{ else } alegere2("SUS, X(n+1), "JOS, X(n-1))$$

Procesul inițial este deci $MM(0)$.

Procesul inițial este deci $MM(0)$.

Dacă P este o funcție reprezentând un proces iar A lista ce conține simbolurile din alfabetul său, funcția LISP

$menu(A,P)$

returnează lista tuturor simbolurilor din A ce pot fi prime evenimente în existența lui P

$menu(A,P) = \text{if } A = \text{NIL} \text{ atunci NIL}$
 then if $P(car(A)) = \text{"BLIP"}$ **then** $menu(cdr(A),P)$
 else $cons(car(A),menu(cdr(A),P))$

Dacă x este în $menu(A,P)$, $P(x)$ nu poate fi "BLIP" și de aceea este o funcție definind comportarea ulterioară a lui P după angajarea în evenimentul x . Dacă y este în $menu(A,P(x))$, atunci $P(x)(y)$ dă comportarea după producerea lui x și y . Se sugerează în acest mod o metodă eficientă de a explora comportarea procesului. Se poate scrie un program care întâi afișează valoarea lui $menu(A,P)$ iar după aceea preia un simbol de la tastatură. Dacă simbolul nu face parte din meniu, se va genera un semnal de eroare și va fi ignorat. Dacă simbolul face parte, el este acceptat iar bucla este reluată prin înlocuirea lui P cu rezultatul aplicării simbolului respectiv lui P . Programul poate fi terminat prin introducerea simbolului "END". Dacă k este lungimea secvenței de simboluri preluate la intrare, funcția ce returnează secvența de ieșiri cerută este

$interact(A,P,k) = cons(menu(A,P),$
 if $car(k) = \text{"END"}$ **then** NIL
 else if $P(car(k)) = \text{"BLIP"}$ **then**
 $cons(\text{"BLIP"},interact(A,P,cdr(k)))$
 else $interact(A,P(car(k)),cdr(k))$

Notățiile folosite mai sus pentru definirea funcțiilor LISP sunt generale și este nevoie de o adaptare a S-expresiiilor pentru o implementare particulară de LISP. De exemplu, în LISPkit funcția prefix va avea forma

$(prefix$
 $lambda$
 $(a\ p)$
 $(lambda\ (x)\ (if\ (eq\ x\ a)\ p\ (quote\ BLIP))))$

Din fericire, în carte vom folosi o mică parte din LISP-ul pur funcțional, astfel că nu vor fi probleme mari cu traducerea și executarea proceselor într-un dialect de pe un anumit calculator.

Dacă aveți la dispoziție mai multe versiuni de LISP, alegeți una cu o corectă legare statică a variabilelor. Un LISP cu o evaluare amânată este de asemenea convenabil din cauza transcrierii directe a ecuațiilor recursive, fără folosirea lui *LABEL*.

AVS=prefix("MON,prefix("CHOC,AVS))

Dacă intrarea și ieșirea sunt implementate prin evaluare progresivă, al treilea parametru al funcției *interact* poate fi tastatura. Meniul pentru procesul *P* va fi primul parametru. Selectând și introducând un simbol din meniurile succesive, un utilizator poate explora interactiv comportarea procesului *P*. În alte versiuni de LISP, funcția *interact* trebuie rescrisă pentru a realiza un efect similar al intrărilor și ieșirilor. Se poate spune că un proces se execută reprezentat fiind printr-o funcție LISP, deci s-a realizat o *implementare* a procesului. La fel, funcția LISP corespunzătoare lui *prefix* operează cu aceste reprezentări, fiind o implementare a operatorului corespunzător dintre procese.

1.5 Urme

O *urmă* a comportării unui proces este o secvență finită de simboluri, înregistrând evenimentele în care procesul s-a angajat până la un anumit moment dat de timp. Ne putem imagina că există un observator cu un carnetel, care urmărește procesele și notează numele fiecărui eveniment ce apare. Cum am presupus deja, ignorăm posibilitatea apariției simultane a evenimentelor. Dacă totuși există și astfel de evenimente, observatorul le va nota într-o ordine oarecare ce nu are importanță.

O urmă va fi definită ca o secvență de simboluri, separate prin virgulă și incluse între *< >*

- <x,y>* constă din două evenimente, *x* urmat de *y*.
- <x>* o secvență ce conține un singur eveniment *x*.
- <>* secvență vidă, fără nici un eveniment.

Exemple

X1 O urmă a automatului simplu de vânzare *AVS* (1.1.2 **X2**) după ce a servit primii doi clienți

<mon,choc,mon,choc>

□

X2 O urmă a aceluiași automat înainte ca al doilea client să-și extragă ciocolata sa

<mon,choc,mon>

Trebuie menționat că atât procesul cât și observatorul nu au noțiunea de tranzație completă. Nerăbdarea sau foamea clientului, precum și posibilitatea automatului de a furniza produse sunt evenimente ce nu figurează în alfabetul proceselor, deci nu pot fi înregistrate, observate. □

X3 Înainte ca un proces să se angajeze în vreun eveniment, carnetelul observatorului este gol. Acest lucru se poate reprezenta prin urma vidă

◇

Orice proces pornește de fapt cu această urmă. □

X4 Automatul de vânzare perfecționat, *AVC* (1.1.3 X4) are următoarele șapte urme de lungime cel mult doi

◇

<i><in2p></i>	<i><in1p></i>
<i><in2p,mare></i>	<i><in2p,mic></i>
<i><inp,in1p></i>	<i><in1p,mic></i>

Evident numai una din cele patru urme de lungime doi apare odată. Alegerea între ele este făcută prin intermediul intențiilor clientului. □

X5 Urma unui *AVC* dacă se ignoră avertismentul introducerii consecutive a trei monezi este

<in1p,in1p,in1p>

Urma nu poate semnala defectarea automatului. Defectarea poate fi detectată prin aceea că, printre atâtea urme posibile nu există nici una de forma

<in1p,in1p,in1p,x>

pentru orice eveniment *x*. □

1.6 Operații cu urme

Urmele joacă un rol principal în înregistrarea, descrierea și înțelegerea comportamentului proceselor. În acest paragraf vom explora câteva din proprietățile generale ale urmelor și operațiile cu ele. Vom folosi următoarele convenții

s, t, u	pentru urme
S, T, U	pentru mulțimi de urme
f, g, h	pentru funcții

1.6.1 Concatenarea

Cea mai importantă operație cu urme este concatenarea, care construiește o urmă din doi operanzi s și t , în această ordine. Rezultatul se va nota

$$s^{\wedge}t$$

De exemplu

$$\begin{aligned} \langle \text{mon}, \text{choc} \rangle^{\wedge} \langle \text{mon}, \text{bonbon} \rangle &= \langle \text{mon}, \text{choc}, \text{mon}, \text{bonbon} \rangle \\ \langle \text{inlp} \rangle^{\wedge} \langle \text{inlp} \rangle &= \langle \text{inlp}, \text{inlp} \rangle \\ \langle \text{inlp}, \text{inlp} \rangle^{\wedge} \diamond &= \langle \text{inlp}, \text{inlp} \rangle \end{aligned}$$

Cea mai importantă proprietate a concatenării este asociativitatea și faptul că secvența \diamond este element unitate

$$\begin{aligned} \text{L1} \quad s^{\wedge} \diamond &= \diamond^{\wedge} s = s \\ \text{L2} \quad s^{\wedge} (t^{\wedge} u) &= (s^{\wedge} t)^{\wedge} u \end{aligned}$$

Următoarele legi sunt atât evidente cât și utile

$$\begin{aligned} \text{L3} \quad s^{\wedge} t &= s^{\wedge} u \equiv t = u \\ \text{L4} \quad s^{\wedge} t &= u^{\wedge} t \equiv s = u \\ \text{L5} \quad s^{\wedge} t &= \diamond \equiv s = \diamond \wedge t = \diamond \end{aligned}$$

Fie f o funcție cu domeniul și codomeniul urme. Funcția se spune că este *strictă* dacă este identitate pentru urma vidă

$$f(\diamond) = \diamond$$

Se spune că f este *distributivă* față de concatenare dacă

$$f(s \wedge t) = f(s) \wedge f(t)$$

Toate funcțiile distributive sunt stricte.

Dacă n este un număr natural, definim t^n ca fiind n copii concatenate ale lui t . Prin inducție după n avem

$$\text{L6} \quad t^0 = \diamond$$

$$\text{L7} \quad t^{n+1} = t \wedge t^n$$

Definiția generează deci cele două legi. Se mai pot deduce două ce pot fi dovedite ușor

$$\text{L8} \quad t^{n+1} = t^n \wedge t$$

$$\text{L9} \quad (s \wedge t)^{n+1} = s \wedge (t \wedge s)^n \wedge t$$

1.6.2 Restricția

Expresia $(t \upharpoonright A)$ semnifică urma t restricționată la simbolurile din mulțimea A . Ea se formează simplu din t omițând simbolurile ce nu fac parte din A . De exemplu

$$\langle pe_loc, sus_jos, pe_loc \rangle \upharpoonright \{sus_jos\} = \langle sus_jos \rangle$$

Restricția este distributivă și de aceea strictă

$$\text{L1} \quad \diamond \upharpoonright A = \diamond$$

$$\text{L2} \quad (s \wedge t) \upharpoonright A = (s \upharpoonright A) \wedge (t \upharpoonright A)$$

Efectul asupra secvenței cu un singur element este evident

$$\text{L3} \quad \langle x \rangle \upharpoonright A = \langle x \rangle \quad \text{dacă } x \in A$$

$$\text{L4} \quad \langle y \rangle \upharpoonright A = \diamond \quad \text{dacă } y \notin A$$

O funcție distributivă este definită unic prin efectul asupra secvenței cu un singur element, deoarece efectele asupra secvențelor de lungimi mai mari pot fi calculate prin aplicarea distributivității fiecărui element și concatenarea rezultatelor. De exemplu, dacă $y \neq x$

$$\begin{aligned} \langle x, y, x \rangle \upharpoonright \{x\} &= (\langle x \rangle \wedge \langle y \rangle \wedge \langle x \rangle) \upharpoonright \{x\} \\ &= (\langle x \rangle \upharpoonright \{x\}) \wedge (\langle y \rangle \upharpoonright \{x\}) \wedge (\langle x \rangle \upharpoonright \{x\}) \\ &= \langle x \rangle \quad \quad \diamond \quad \quad \langle x \rangle \\ &= \langle x, x \rangle \end{aligned}$$

din L2

din L3, L4

Următoarele legi ne arată relația dintre restricție și operațiile cu mulțimi. O urmă restricționată la o mulțime vidă ne dă o secvență vidă. O restricție succesivă la două mulțimi are același efect ca și restricția la intersecția mulțimilor. Aceste două legi pot fi demonstrate prin inducție după lungimea secvențelor.

$$L5 \quad s \upharpoonright \{\} = \langle \rangle$$

$$L6 \quad (s \upharpoonright A) \upharpoonright B = s \upharpoonright (A \cap B)$$

1.6.3 Cap și coadă

Dacă s este o secvență nevidă, primul ei simbol se va nota s_0 , iar rezultatul eliminării primului simbol se va nota s' . De exemplu

$$\langle x, y, x \rangle_0 = x$$

$$\langle x, y, x \rangle' = \langle y, x \rangle$$

Ambele operații sunt inoperante pe secvența vidă

$$L1 \quad (\langle x \rangle^s)_0 = x$$

$$L2 \quad (\langle x \rangle^s)' = s$$

$$L3 \quad s = (\langle s_0 \rangle^s)' \quad \text{dacă } s \neq \langle \rangle$$

Următoarea lege dă o metodă comodă de a arăta că două urme sunt identice

$$L4 \quad s = t \Leftrightarrow (s = \langle \rangle \vee (s_0 = t_0 \wedge s' = t'))$$

1.6.4 Stea

Mulțimea A^* este mulțimea tuturor secvențelor finite (inclusiv $\langle \rangle$) ce se formează din simbolurile mulțimii A . Când astfel de urme sunt restricționate la mulțimea A , ele rămân nemodificate. Această proprietate permite o definiție simplă

$$A^* = \{s \mid s \upharpoonright A = s\}.$$

Următoarele legi sunt consecințe ale definiției

$$L1 \quad \langle \rangle \in A^*$$

$$L2 \quad \langle x \rangle \in A^* \Leftrightarrow x \in A$$

$$L3 \quad (s^i) \in A^* \Leftrightarrow s \in A^* \wedge i \in A^*$$

Ele sunt suficient de puternice pentru a determina dacă o urmă este sau nu un membru al lui A^* . De exemplu, dacă $x \in A$ și $y \in A$

$$\begin{aligned} \langle x, y \rangle \in A^* &= (\langle x \rangle \wedge \langle y \rangle) \in A^* \\ &= (\langle x \rangle \in A^*) \wedge (\langle y \rangle \in A^*) && \text{din L3} \\ &= \text{true} \wedge \text{false} && \text{din L2} \\ &= \text{false} \end{aligned}$$

Altă lege utilă ar putea fi definirea recursivă a lui A^*

$$\text{L4} \quad A^* = \{t \mid t = \diamond \vee (t_0 \in A \wedge t' \in A^*)\}.$$

1.6.5 Ordonarea

Dacă s este o subsecvență inițială a lui t , este posibil de a găsi o extensie u a lui s astfel ca $s \wedge u = t$. De aceea definim o relație de ordine

$$s \leq t = (\exists u. s \wedge u = t)$$

și spunem că s este un *prefix* a lui t . De exemplu

$$\begin{aligned} \langle x, y \rangle &\leq \langle x, y, x, w \rangle \\ \langle x, y \rangle &\leq \langle z, y, x \rangle = x = z \end{aligned}$$

Relația \leq este o ordine parțială, cu cel mai mic element \diamond , cum rezultă din legile L1-L4

L1	$\diamond \leq s$	cel mai mic element
L2	$s \leq s$	reflexivitate
L3	$s \leq t \wedge t \leq s \Rightarrow s = t$	antisimetrie
L4	$s \leq t \wedge t \leq u \Rightarrow s \leq u$	tranzitivitate

Următoarea lege împreună cu L1 face posibil verificarea ordinii $s \leq t$

$$\text{L5} \quad (\langle x \rangle \wedge s) \leq t = t \neq \diamond \wedge x = t_0 \wedge s \leq t'$$

Prefixele unei secvențe date sunt total ordonate

$$\text{L6} \quad s \leq u \wedge t \leq u \Rightarrow s \leq t \vee t \leq s$$

Dacă s este o subsecvență a lui t (nu-i necesar inițială), spunem că avem relația s în t . Aceasta se poate defini astfel

$$s \text{ in } t \equiv (\exists u, v. t = u \wedge s \wedge v)$$

Această relație este de asemenea o ordine parțială, deoarece satisface legile L1-L4 de mai sus. Relația mai satisface și legea

$$L7 \quad (\langle x \rangle \wedge s) \text{ in } t \equiv t \neq \diamond \wedge (t_0 = x \wedge s \leq t') \vee (\langle x \rangle \wedge s) \text{ in } t'$$

O funcție f cu domeniul și codomeniul mulțimi de urme se spune că este *monotonă* dacă respectă ordinea \leq

$$f(s) \leq f(t) \quad \text{dacă } s \leq t$$

Toate funcțiile distributive sunt monotone, de exemplu

$$L8 \quad s \leq t \Rightarrow (s \upharpoonright A) \leq (t \upharpoonright A)$$

O funcție de două argumente poate fi monotonă în oricare din ele separat, celălalt fiind ținut constant. De exemplu, concatenarea este monotonă în al doilea argument (dar nu în primul)

$$L9 \quad t \leq u \Rightarrow (s \wedge t) \leq (s \wedge u)$$

O funcție monotonă în ambele argumente se numește pur și simplu monotonă.

1.6.6 Lungime

Lungimea unei urme t este notată $\#t$. De exemplu

$$\# \langle x, y, x \rangle = 3$$

Legile care definesc $\#$ sunt

$$L1 \quad \# \diamond = 0$$

$$L2 \quad \# \langle x \rangle = 1$$

$$L3 \quad \# \langle s \wedge t \rangle = (\#s) + (\#t)$$

Numărul aparițiilor simbolilor din A în secvența t este dată de $\#(t \upharpoonright A)$.

$$L4 \quad \#(t \upharpoonright (A \cup B)) = \#(t \upharpoonright A) + \#(t \upharpoonright B) - \#(t \upharpoonright (A \cap B))$$

$$L5 \quad s \leq t \Rightarrow \#s \leq \#t$$

$$L6 \quad \#(t^n) = n \times (\#t)$$

Numărul aparițiilor simbolului x în urma s este definită de

$$s \downarrow x = \#(s \uparrow \{x\})$$

1.7 Implementarea urmelor

În vederea implementării urmelor și a operațiilor cu ele, folosim limbajul de nivel înalt pentru prelucrarea listelor, foarte potrivit pentru aceasta, LISP. Urmele vor fi reprezentate prin liste de atomi corespunzători evenimentelor

$$\diamond = NIL$$

$$\langle mon \rangle = cons("MON, NIL)$$

$$\langle mon, choc \rangle = ("MON\ CHOC)$$

$$\text{adică } cons("MON, cons("CHOC, NIL))$$

Operațiile cu urme pot fi ușor implementate cu ajutorul funcțiilor de liste

$$t_0 = car(t)$$

$$t' = cdr(t)$$

$$\langle x \rangle^s = cons(x, s)$$

Concatenarea este implementată cu ajutorul binecunoscutei funcții *append*, definită recursiv

$$s^t = append(s, t)$$

unde $append(s, t) = \text{if } s = NIL \text{ then } t \text{ else } cons(car(s), append(cdr(s), t))$

corectitudinea definiției decurge din legile

$$\diamond^t = t$$

$$s^t = \langle s_0 \rangle^{\wedge} (s'^t) \quad \text{dacă } s \neq \diamond$$

Terminarea funcției LISP *append* este garantată de faptul că lista din primul argument al fiecărui apel recursiv este mai scurtă decât în precedentul apel. Datorită unor raționamente analoage se poate stabili corectitudinea implementărilor altor operații definite mai jos.

Pentru a implementa restricția, folosim o listă B drept mulțimea elementelor. Testul $(x \in B)$ se realizează prin apelul funcției

$$estemembru(x, B) = \text{if } B = NIL \text{ then } false$$

else if $x = \text{car}(B)$ then true
 else $\text{estemembru}(x, \text{cdr}(B))$

$(s \upharpoonright A)$ poate fi implementat prin funcția

$\text{restrict}(s, B) =$ if $s = \text{NIL}$ then NIL
 else if $\text{estemembru}(\text{car}(s), B)$
 then $\text{cons}(\text{car}(s), \text{restrict}(\text{cdr}(s), B))$
 else $\text{restrict}(\text{cdr}(s), B)$

Testul $(s \upharpoonright t)$ se implementează cu o funcție care returnează *true* sau *false*.
 Ne bazăm pe 1.6.5 L1 și L5

$\text{esteprefix}(s, t) =$ if $s = \text{NIL}$ then true
 else if $t = \text{NIL}$ then false
 else $\text{car}(s) = \text{car}(t) \wedge \text{esteprefix}(\text{cdr}(s), \text{cdr}(t))$

1.8 Urmele proceselor

În paragraful 1.6 urma unui proces a fost introdusă ca fiind înregistrarea secvențială a comportării procesului până la un moment dat. Înainte de începerea procesului nu se poate ști care din urmele sale posibile va fi urmată. Alegerea este independentă de controlul procesului, fiind determinată de factorii de mediu. Este clar că mulțimea tuturor urmelor posibile ale unui proces P poate fi cunoscută apriori, astfel că putem defini $\text{urme}(P)$ pentru a o defini.

Exemple

X1 Singura urmă ce rezultă din comportarea procesului *STOP* este \diamond .
 Carnețelul observatorului în cazul acestui proces rămâne gol totdeauna

$\text{urme}(\text{STOP}) = \{\diamond\}$ □

X2 Există numai două urme ale automatului de vânzare în care se introduce o monedă și apoi se defectează

$\text{urme}(\text{mon} \rightarrow \text{STOP}) = \{\diamond, \langle \text{mon} \rangle\}$ □

X3 Un ceas care ticăie mereu

$\text{urme}(\mu X. \text{tic} \rightarrow X) = \{\diamond, \langle \text{tic} \rangle, \langle \text{tic}, \text{tic} \rangle, \dots\}$
 $= \{\text{tic}\}^*$

La cele mai multe din procese, mulțimea urmelor este infinită, cu toate că fiecare urmă în parte este finită. \square

X4 Un automat simplu de vânzare \square

$$\text{urme}(\mu X. \text{mon} \rightarrow \text{choc} \rightarrow X) = \{s \mid \exists n. s \leq \langle \text{mon}, \text{choc} \rangle^n\}$$

1.8.1 Legi

În acest paragraf vom deduce mulțimea urmelor unui proces dat folosind notațiile introduse. Cum s-a arătat mai sus, *STOP* are o singură urmă

$$\text{L1} \quad \text{urme}(\text{STOP}) = \{t \mid t = \diamond\} = \{\diamond\}$$

Urma lui $(c \rightarrow P)$ poate fi vădă, deoarece \diamond este urma comportării oricărui proces până la momentul când se angajează în prima acțiune. Astfel, orice urmă nevidă a lui P începe cu c și coada este o posibilă urmă a procesului.

$$\text{L2} \quad \text{urme}(c \rightarrow P) = \{t \mid t = \diamond \vee (t_0 = c \wedge t' \in \text{urme}(P))\} = \{\diamond\} \cup \{ \langle c \rangle \wedge t' \mid t' \in \text{urme}(P) \}$$

Urma comportării unui proces care oferă alegere inițială între evenimente trebuie să fie urma uneia din alternative

$$\text{L3} \quad \text{urme}(c \rightarrow P \mid d \rightarrow P) = \{t \mid t = \diamond \vee (t_0 = c \wedge t' \in \text{urme}(P)) \vee (t_0 = d \wedge t' \in \text{urme}(Q))\}$$

Aceste trei legi pot fi sintetizate într-o singură lege, mai generală, despre alegere

$$\text{L4} \quad \text{urme}(x: B \rightarrow P(x)) = \{t \mid t = \diamond \vee (t_0 \in B \wedge t' \in \text{urme}(P(t_0)))\}$$

Ceva mai complicat este de a afla mulțimea urmelor unui proces definit recursiv. Un astfel de proces este soluția unei ecuații de forma

$$X = F(X)$$

Mai întâi, definim recursivitatea printr-o inducție

$$\begin{aligned} F^0(X) &= X \\ F^{n+1}(X) &= F(F^n(X)) \\ &= F^n(F(X)) \end{aligned}$$

$$= \underbrace{F(\dots(F(F(X)))\dots)}_{n \text{ ori}}$$

Apoi, dacă F este cu gardă efectivă, putem defini

$$\text{L5} \quad \text{urme}(\mu X:A. F(X)) = \bigcup_{n \geq 0} \text{urme}(F^n(\text{STOP}_A))$$

Exemple

X1 Să ne reamintim cum am definit RUN_A în 1.1.3 X8

$$\mu X:A. F(X)$$

unde $F(X) = (x:A \rightarrow X)$

Dorim să demonstrăm că

$$\text{urme}(RUN_A) = A^*$$

$$\text{Demonstrație. } A^* = \bigcup_{n \geq 0} \{s \mid s \in A^* \wedge \#s \leq n\}$$

Folosind L5 este suficient de demonstrat pentru orice n că

$$\text{urme}(F^n(\text{STOP}_A)) = \{s \mid s \in A^* \wedge \#s \leq n\}$$

Acest lucru îl vom face prin inducție după n

(1) pentru $n=0$

$$\begin{aligned} \text{urme}(\text{STOP}_A) &= \{\langle \rangle\} \\ &= \{s \mid s \in A^* \wedge \#s \leq 0\} \end{aligned}$$

(2) $\text{urme}(F^{n+1}(\text{STOP}_A))$

$$\begin{aligned} &= \text{urme}(x:A \rightarrow F^n(\text{STOP}_A)) \\ &= \{t \mid t = \langle \rangle \vee (t_0 \in A \wedge t' \in \text{urme}(F^n(\text{STOP}_A)))\} \\ &= \{t \mid t = \langle \rangle \vee (t_0 \in A \wedge (t' \in A^* \wedge \#t' \leq n))\} \\ &= \{t \mid (t = \langle \rangle \vee (t_0 \in A \wedge t' \in A^*)) \wedge \#t \leq n+1\} \\ &= \{t \mid t \in A^* \wedge \#t \leq n+1\} \end{aligned}$$

definiția lui F , F^{n+1}

L4

ipoteza de inducție
proprietatea lui #

1.6.4 L4 \square

X2 Vrem să demonstrăm corectitudinea lui 1.8 X4, adică

$$\text{urme}(AVS) = \bigcup_{n \geq 0} \{s \mid s \leq \langle \text{mon}, \text{choc} \rangle^n\}$$

Demonstrație. Propoziția de inducție este

$$\text{urme}(F^n(AVS)) = \{t \mid t \leq \langle \text{mon}, \text{choc} \rangle^n\}$$

unde $F(X) = (\text{mon} \rightarrow \text{choc} \rightarrow X)$

- (1) $\text{urme}(STOP) = \{\diamond\} = \{s \mid s \leq \langle \text{mon}, \text{choc} \rangle^n\}$ 1.6.1 L6
- (2) $\text{urme}(\text{mon} \rightarrow \text{choc} \rightarrow F^n(STOP))$
 $= \{\diamond, \langle \text{mon} \rangle\} \cup \{\langle \text{mon}, \text{choc} \rangle^t \mid t \in \text{urme}(F^n(STOP_A))\}$ aplic L2
 de două ori
 $= \{\diamond, \langle \text{mon} \rangle\} \cup \{\langle \text{mon}, \text{choc} \rangle^t \mid t \leq \langle \text{mon}, \text{choc} \rangle^n\}$ ipoteza de inducție
 $= \{s \mid s = \diamond \vee (s = \langle \text{mon} \rangle \vee \exists t. s = \langle \text{mon}, \text{choc} \rangle^t \wedge t \leq \langle \text{mon}, \text{choc} \rangle^n)\}$
 $= \{s \mid s \leq \langle \text{mon}, \text{choc} \rangle^{n+1}\}$

Concluzia decurge din L5 □

Cum s-a menționat în paragraful 1.5, o urmă este o secvență de simboluri înregistrând evenimentele în care s-a angajat procesul P până la un moment dat de timp. Astfel, rezultă clar că \diamond este urma oricărui proces până la momentul când se angajează în primul eveniment. Mai mult, dacă s^t este urma unui proces până la un moment de timp, atunci s este urma până la un moment anterior. În fine, orice eveniment care apare trebuie să fie în alfabetul procesului considerat. Aceste trei concluzii sunt formalizate în legile următoare

- L6 $\diamond \in \text{urme}(P)$
 L7 $s^t \in \text{urme}(P) \Rightarrow s \in \text{urme}(P)$
 L7 $\text{urme}(P) \subseteq (\alpha P)^*$

Există o strânsă legătură între urmele unui proces și reprezentarea sub formă de arbore a comportării sale. Urma comportării procesului până la un moment de timp, corespunzător unui nod al arborelui, este secvența etichetelor evenimentelor pe calea de la rădăcină la acel nod. De exemplu, pentru arborele corespunzător lui AVC , fig. 1.1, urma obținută prin parcurgerea căii de la rădăcină la nodul punctat plin este

$\langle \text{in2p}, \text{mic}, \text{exlp} \rangle$

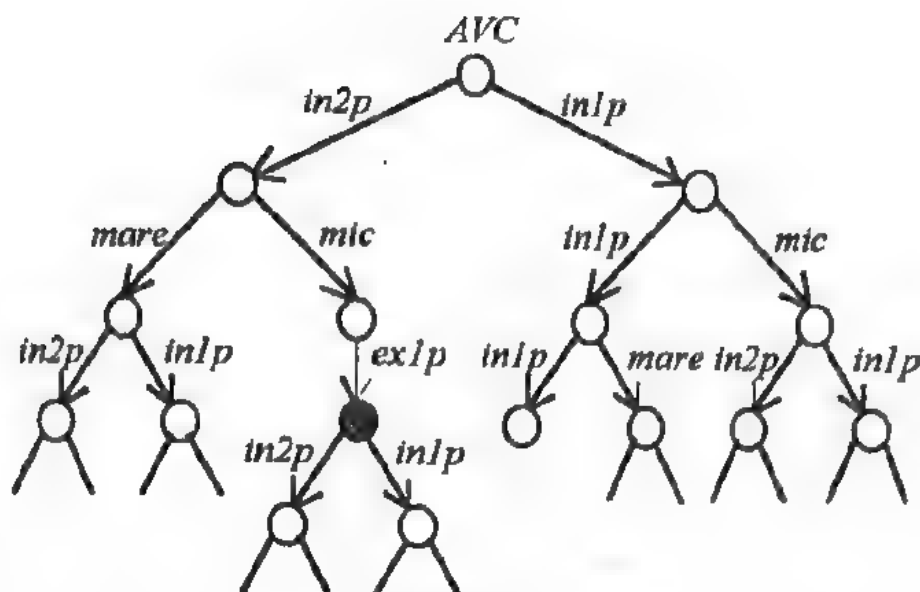


Figura 1.1

Căile sau subcăile din arbore sunt urme pentru procesul dat. Acest lucru este concluzionat în L7. Legea L6 semnifică o urmă vidă pentru calea de la nodul rădăcină la el însuși. Ultima lege, L8, înseamnă că *urme*-le unui proces sunt incluse în mulțimea căilor ce pleacă din rădăcină.

Reciproc, deoarece arcele incidente exterior oricărui nod sunt etichetate cu evenimente diferite, orice urmă corespunde unei singure căi de la nodul rădăcină la un nod oarecare. Astfel, orice mulțime de urme satisfăcând L6 și L7 este o reprezentare matematică convenabilă a unui arbore cu arcele incidente exterior fiecărui nod etichetate cu evenimente diferite.

1.8.2 Implementare

Presupunem că un proces este implementat cu ajutorul unei funcții LISP P și fie s o urmă. Putem testa dacă s este o posibilă urmă a lui P cu funcția

$esteurmă(s, P) = \text{if } s = \text{NIL} \text{ then true}$
 $\quad \text{else if } P(s_0) = \text{"BLIP"} \text{ then false else } esteurmă(s', P(s_0))$

Când s este finită, generată prin explorarea unei părți a comportării procesului P , recursivitatea se termină.

1.8.3 După

Dacă $s \in urme(P)$ atunci

$P/s \qquad (P \text{ după } s)$

este un proces ce se comportă identic cu P după momentul când P s-a angajat în toate acțiunile înregistrate de urma s . Dacă s nu este o urmă a lui P , (P/s) nu este definit.

Exemple

X1 $(AVS/\langle mon \rangle) = (choc \rightarrow AVS)$ □

X2 $(AVS/\langle mon, choc \rangle) = AVS$ □

X3 $(AVC/\langle mon \rangle^3) = STOP$ □

X4 Pentru evitarea pierderii unei ciocolate, generată de instalarea lui $AVMMC$ (1.1.3 X5, X6), proprietarul aparatului decide să mănânce el ciocolata

$(AVMCCM/\langle choc \rangle) = AVS2$ □

În reprezentarea arborescentă a lui P (fig. 1.1), (P/s) semnifică subarborele cu rădăcina în nodul terminal al căii s . Astfel, subarborele corespunzător nodului plin ca rădăcină este

$(AVC/\langle in2p, mic, ex1p \rangle)$

Legile următoare descriu semnificația operatorului $/$. Dacă nu execută nimic, un proces rămâne neschimbat

L1 $P/\diamond = P$

Comportarea lui P după angajarea în s^*t este aceeași cu a lui (P/s) după angajarea în t .

L2 $P/(s^*t) = (P/s)/t$

După angajarea într-un eveniment singular c , comportarea procesului este definită de această alegere inițială

L3 $(x:B \rightarrow P(x))/\langle c \rangle = P(c)$ dacă $c \in B$

Se observă, ca o consecință, că $\langle c \rangle$ este inversul lui $c \rightarrow$

L3A $(c \rightarrow P)/\langle c \rangle = P$

Urmele lui (P/s) le definim astfel

$$L4 \quad urme(P/s) = \{t \mid s^{\wedge}t \in urme(P)\} \quad \text{dacă } s \in urme(P)$$

Pentru a arăta că un proces nu se oprește niciodată, este suficient să arătăm că

$$P/s \neq STOP \quad \text{pentru toate } s \in urme(P)$$

O altă proprietate binecunoscută a proceselor este *ciclicitatea*. Un proces este definit ca ciclic dacă din orice situație este posibil să se reîntoarcă în starea inițială

$$\forall s. urme(P). \exists t. (P/(s^{\wedge}t) = P)$$

$STOP$ este un proces ciclic banal. Orice alt proces ciclic are proprietatea că nu se oprește niciodată.

Exemple

X1 Următoarele procese sunt ciclice, (1.1.3 X8, 1.1.2 X2, 1.1.3 X3, 1.1.4 X2)

$$RUN_A, AFS, (choc \rightarrow AFS), AICB, MM_7 \quad \square$$

X2 Următoarele procese nu sunt ciclice, deoarece nu este posibilă revenirea în starea inițială (1.1.2 X2, 1.1.3 X3, 1.1.3 X2)

$$(mon \rightarrow AFS), (choc \rightarrow AICB), (pe_loc \rightarrow MM_7)$$

De exemplu, starea inițială a lui $(choc \rightarrow AICB)$ presupune obținerea unei singure ciocolate pe când stările ulterioare presupun o alegere între *choc* și *bunbon*, ceea ce evident este diferit. \square

Atenție. Utilizarea lui / într-un proces definit recursiv are consecința nefastă de a invalida gărzile, generându-se pericolul unor soluții multiple pentru ecuații. De exemplu

$$X = (a \rightarrow (X / \langle a \rangle))$$

este fără gardă și soluția poate fi orice proces de forma

$$a \rightarrow P$$

oricare ar fi P .

Demonstrație. $(a \rightarrow ((a \rightarrow P) / \langle a \rangle)) = (a \rightarrow P)$ din L3A

Din acest motiv nu vom folosi operatorul / în definirea proceselor recursive.

1.9 Alte operații cu urme

Acest paragraf prezintă alte câteva operații cu urme. Pe moment paragraful poate fi parcurs opțional deoarece vor fi făcute referiri la aceste operații în capitolele următoare.

1.9.1 Schimbare de simbol

Fie f o funcție definită pe o mulțime A de simboluri și cu valori într-o mulțime B . Putem obține din f o funcție f^* , definită pe mulțimea A^* și cu valori în B^* , mulțimi de secvențe, aplicând f fiecărui element din secvența A . De exemplu, *dublu* este o funcție ce își dublează argumentul întreg

$$\text{dublu}^*(\langle 1, 5, 3, 1 \rangle) = \langle 2, 10, 6, 2 \rangle$$

O funcție stea este evident distributivă și de aceea strictă

$$\text{L1 } f^*(\diamond) = \diamond$$

$$\text{L2 } f^*(\langle x \rangle) = \langle f(x) \rangle$$

$$\text{L3 } f^*(s \wedge t) = f^*(s) \wedge f^*(t)$$

Alte legi sunt consecințe evidente

$$\text{L4 } f^*(s)_0 = f(s_0) \quad \text{dacă } s \neq \diamond$$

$$\text{L5 } \#f^*(s) = \#s$$

Prezentăm mai jos o lege care pare evidentă, dar nu este adevărată în toate cazurile

$$f^*(s \upharpoonright A) = f^*(s) \upharpoonright f(A)$$

unde $f(A) = \{f(x) | x \in A\}$

Un contraexemplu este dat de funcția f unde

$$f(b)=f(b)=c$$

unde $b \neq c$

Astfel $f^*(\langle b \rangle \upharpoonright \{c\}) = f^*(\langle \rangle)$

deoarece $b \neq c$

$$= \langle \rangle$$

L1

$$\neq \langle c \rangle$$

$$= \langle c \rangle \upharpoonright \{c\}$$

$$= f^*(\langle b \rangle) \upharpoonright f(\{c\})$$

deoarece $f(c)=c$

Totuși legea este adevărată dacă f este injectivă

L6 $f^*(s \upharpoonright A) = f^*(s) \upharpoonright f(A)$

dacă f este injectivă

1.9.2 Unificare

Fie s o secvență ale cărei elemente sunt de asemenea secvențe. Vom spune că $\wedge s$ este unificarea tuturor elementelor într-o ordine naturală. De exemplu

$$\wedge \langle \langle 1,3 \rangle, \langle \rangle, \langle 7 \rangle \rangle = \langle 1,3 \rangle \wedge \langle \rangle \wedge \langle 7 \rangle = \langle 1,3,7 \rangle$$

Acest operator este distributiv

L1 $\wedge \langle \rangle = \langle \rangle$

L2 $\wedge \langle s \rangle = s$

L3 $\wedge s \wedge t = (\wedge s) \wedge (\wedge t)$

1.9.3 Întreșeserea

O secvență s se spune că este o întreșesere a secvențelor t și u dacă poate fi descompusă într-o serie de subsecvențe aparținând lui t și u .

$$s = \langle 1,6,3,1,5,4,2,7 \rangle$$

este o întreșesere a lui t și u , unde

$$t = \langle 1,6,5,2,7 \rangle \quad \text{și} \quad u = \langle 3,1,4 \rangle$$

O definiție recursivă a întreșeserii poate fi dată cu ajutorul următoarelor legi

L1 $\langle \rangle \text{ întreșese } (t,u) = (t = \langle \rangle \wedge u = \langle \rangle)$

L2 $s \text{ întreșese } (t,u) = s \text{ întreșese } (u,t)$

$$\text{L3} \quad (\langle x \rangle^s) \text{întreşese}(t, u) \equiv (t \neq \diamond \wedge t_0 = x \wedge s \text{întreşese}(t', u)) \\ \vee (u \neq \diamond \wedge u_0 = x \wedge s \text{întreşese}(t, u'))$$

1.9.4 Indexarea

Dacă $0 \leq i < \#s$, folosim notația $s[i]$ pentru al i -lea element al secvenței s , cu definiția dată de L1

$$\begin{array}{ll} \text{L1} & s[0] = s_0 \wedge s[i+1] = s'[i] \quad \text{dacă } s \neq \diamond \\ \text{L2} & (f^*(s))[i] = f(s[i]) \quad \text{pentru } i < \#s \end{array}$$

1.9.5 Reversul

Dacă s este o secvență, \overline{s} este secvența ce rezultă prin inversarea ordinii elementelor. De exemplu

$$\overline{\langle 3, 5, 37 \rangle} = \langle 37, 5, 3 \rangle$$

Reversul este complet definit de următoarele legi

$$\begin{array}{ll} \text{L1} & \overline{\langle \rangle} = \diamond \\ \text{L2} & \overline{\langle x \rangle} = \langle x \rangle \\ \text{L3} & \overline{s \wedge t} = \overline{t} \wedge \overline{s} \end{array}$$

Reversul se bucură de o serie de proprietăți algebrice

$$\text{L4} \quad \overline{\overline{s}} = s$$

Descoperirea altor proprietăți poate fi făcută de către cititor. Un rezultat util referitor la revers ne spune că $\overline{s_0}$ este ultimul element al unei secvențe, sau mai general

$$\text{L5} \quad \overline{s}[i] = s[\#s - i - 1] \quad \text{pentru } i < \#s$$

1.9.6 Selecția

Dacă s este o secvență de perechi, definim $s \downarrow x$ ca fiind rezultatul selectării din s a tuturor perechilor cu prim element x și înlocuirii lor cu al doilea element. Perechea va conține un punct între cele două elemente. Astfel avem

$$s = \langle a.7, b.9, a.8, c.0 \rangle$$

$$\begin{array}{ll} \text{atunci} & s \downarrow a = \langle 7, 8 \rangle \\ \text{iar} & s \downarrow d = \langle \rangle \end{array}$$

$$L1 \quad \langle \rangle \downarrow x = \langle \rangle$$

$$L2 \quad (\langle y.z \rangle \wedge t) \downarrow x = t \downarrow x$$

$$\text{dacă } y \neq z$$

$$L3 \quad (\langle x.z \rangle \wedge t) \downarrow x = \langle z \rangle \wedge (t \downarrow x)$$

Dacă s nu este o secvență de perechi, $s \downarrow a$ semnifică numărul aparițiilor lui a în s (așa cum rezultă din paragraful 1.6.6).

1.9.7 Compunerea

Notăm cu simbolul \checkmark un eveniment ce reprezintă terminarea cu succes a unui proces angajat în el. O primă concluzie este că acest simbol poate apărea doar la sfârșitul unei urme. Fie t o urmă ce reprezintă înregistrarea secvenței evenimentelor care încep după ce s-a terminat s cu succes. Compunerea lui s cu t este notată $(s; t)$. Dacă \checkmark nu apare în s , atunci nici t nu poate începe

$$L1 \quad s; t = s \quad \text{dacă } \neg(\langle \checkmark \rangle \text{ în } s)$$

Dacă \checkmark este sfârșitul lui s , el este eliminat și t se adaugă rezultatului

$$L2 \quad (s^{\wedge} \langle \checkmark \rangle); t = s^{\wedge} t \quad \text{dacă } \neg(\langle \checkmark \rangle \text{ în } s)$$

Simbolul poate fi privit ca un conector de unire al secvențelor s și t . În absența lui din s , secvența t nu poate fi adăugată (L1). Dacă totuși \checkmark apare în interiorul unei urme (incorect), stipulăm, pentru compatibilitate, că toate simbolurile după el sunt irelevante și deci trebuie omise

$$L2A \quad (s^{\wedge} \langle \checkmark \rangle^{\wedge} u); t = s^{\wedge} t \quad \text{dacă } \neg(\langle \checkmark \rangle \text{ în } s)$$

Acest fel mai special de compunere se bucură de un număr de proprietăți algebrice. Ca și unificarea, compunerea este asociativă. Spre deosebire de unificare, compunerea este monotonă atât în primul cât și în al doilea argument. De asemenea, este strict monotonă în primul argument și are (\checkmark) ca element unitate la stânga

$$L3 \quad s; (t; u) = (s; t); u$$

$$L4A \quad s \leq t \Rightarrow ((u; s) \leq (u; t))$$

$$L4B \quad s \leq t \Rightarrow ((s; u) \leq (t; u))$$

L5 $\langle \rangle; t = \langle \rangle$

L6 $\langle \checkmark \rangle; t = t$

Dacă \checkmark apare la sfârșitul urmei, $\langle \checkmark \rangle$ poate fi membrul drept al operatorului compunere

L7 $s; \langle \checkmark \rangle = s$

dacă $\neg(\langle \checkmark \rangle \ln (\bar{s}))$

1.10 Specificații

Specificația pentru un produs este descrierea comportării sale. Descrierea este de fapt un predicat ale cărui variabile independente corespund unui aspect observabil al comportării sale. De exemplu, specificarea unui amplificator electronic cu intrarea în gama un volt și factorul de câștig aproximativ 10 este dată de predicatul

$$AMP10 = (0 \leq v \leq 1 \Rightarrow |v' - 10 * v| \leq 1)$$

În această specificare, v semnifică tensiunea de intrare iar v' tensiunea de ieșire. O semnificație clară a variabilelor este esențială la utilizarea matematicii în alte științe și inginerie.

În cazul unui proces, cea mai importantă și relevantă observație a comportării sale este urma evenimentelor până la un moment de timp. Vom folosi variabila ur pentru a nota o urmă arbitrară a procesului specificat, așa cum v și v' semnificau observațiile privind tensiunile, în exemplul anterior.

Exemple

X1 Patronul unui automat de vânzare nu vrea să aibă vreo pierdere la instalare. De aceea el cere în specificație ca numărul ciocolatelor furnizate să nu depășească pe cel al monedelor introduse în aparat

$$FĂRĂPIERDERI = (\#(ur \upharpoonright \{choc\}) \leq \#(ur \upharpoonright \{mon\}))$$

□

În continuare vom folosi abrevierea (introdusă în 1.6.6)

$$ur \downarrow c = \#(ur \upharpoonright \{c\})$$

pentru a nota numărul aparițiilor lui c în ur .

X2 Clientul unui automat de vânzare vrea să se asigure că nu va trebui să introducă noi monezi până când nu primește ciocolatele plătite

$$ECHIT1 = ((ur \downarrow mon) \leq (ur \downarrow choc) + 1) \quad \square$$

X3 Producătorul unui automat de vânzare trebuie să satisfacă cerințele atât ale patronilor cât și ale clienților

$$SPEC_{AV} = F\ddot{A}R\ddot{A}PIERDERI \wedge ECHIT1 = (0 \leq ((ur \downarrow mon) - (ur \downarrow choc)) \leq 1) \quad \square$$

X4 O specificație a funcționării corecte a unui automat interzice introducerea a trei monezi de un penny succesiv

$$CORECT_{AVC} = (\neg (<in\ 1p>^3 \text{ in } ur)) \quad \square$$

X5 Specificația pentru un automat sigur

$$SIGUR_{AVC} = (ur \in urme(AVC) \wedge CORECT_{AVC}) \quad \square$$

X6 Specificația pentru *AVS2* (1.1.3 X6)

$$0 \leq ((ur \downarrow mon) - (ur \downarrow choc)) \leq 2 \quad \square$$

1.10.1 Satisfacerea

Dacă P este un produs care îndeplinește specificația S spunem că P *satisfacă* S , notat

$$P \text{ sat } S$$

Satisfacerea înseamnă că orice observație posibilă a comportării lui P este descrisă de S . Cu alte cuvinte, S este adevărată pentru orice valori date variabilelor, valori generate din observațiile asupra lui P , sau mai formal

$$\forall ur. ur \in urme(P) \Rightarrow S$$

De exemplu, următorul tabel prezintă observații referitoare la proprietățile unui amplificator

	1	2	3	4	5
v	0	.5	.5	2	.1
v'	0	5	4	1	3

Toate observațiile cu excepția ultimelor sunt descrise de *AMP10*. A doua și a treia coloană arată faptul că ieșirea amplificatorului nu este complet determinată de intrare. A patra coloană arată că dacă tensiunea de intrare nu este din gama specificată, tensiunea de ieșire poate avea orice valoare fără a contrazice specificațiile. (În acest exemplu simplu s-a ignorat posibilitatea ca o tensiune de intrare prea mare să distrugă amplificatorul.)

Următoarele legi dau proprietățile cele mai generale referitoare la relația *satisfacă*. Specificația *true*, care presupune inexistența vreunei constrângeri indiferent de observațiile referitoare la produs, este satisfăcută de orice produs. Putem spune că orice produs defect satisfacă această cea mai slabă specificație

L1 $P \text{ sat } true$

Dacă un produs satisfacă două specificații diferite, atunci satisfacă și conjuncția lor

L2A Dacă $P \text{ sat } S$
și $P \text{ sat } T$
atunci $P \text{ sat } (S \wedge T)$

Legea L2A se poate generaliza la un număr infinit de conjuncții prin cuantificare. Fie $S(n)$ un predicat cu variabila n

L2 Dacă $\forall n. (P \text{ sat } S(n))$
atunci $P \text{ sat } (\forall n. S(n))$

dacă P nu conține pe n .

Dacă există o implicație logică între specificațiile S și T , atunci orice observație descrisă de S este de asemenea descrisă de T . Totodată, orice produs care satisfacă S trebuie să satisfacă și specificația mai slabă T

L3 Dacă $P \text{ sat } S$
și $S \Rightarrow T$
atunci $P \text{ sat } T$

Prin prisma ultimei legi vom scrie în cazul unor reguli, în care intervin implicații de forma $S \Rightarrow T$

$P \text{ sat } S$
 $\Rightarrow T$

ca o prescurtare pentru

$$\begin{array}{l} P \text{ sat } S \\ S \Rightarrow T \\ P \text{ sat } T \end{array}$$

din L3

Legile și explicațiile date mai sus pot fi aplicate oricărui produs și oricărei specificații. În următorul paragraf se vor prezenta legile specifice proceselor.

1.10.2 Reguli

În proiectarea unui produs trebuie să ne asigurăm că acesta va satisface specificațiile. Responsabilitatea poate fi asumată folosind raționamente din diverse domenii ale matematicii, ca de exemplu geometria diferențială, calculul diferențial și integral. În prezentul paragraf se prezintă un set de legi ce folosesc raționamente algebrice, logice, permițând demonstrarea îndeplinirii specificației S de către procesul P .

Vom scrie $S(ur)$ pentru a arăta că specificația are pe ur drept variabilă liberă. Un alt motiv pentru care ur este explicită este posibilitatea de a o substitui cu expresii complicate, ca de exemplu ur'' . Este evident că atât S cât și $S(ur)$ au și alte variabile în afară de ur .

Orice observație relativă la procesul $STOP$ va fi întotdeauna urma vidă, deoarece procesul nu face nimic

LAA $STOP \text{ sat } (ur = \diamond)$

Urma procesului $(c \rightarrow P)$ este inițial vidă. Orice urmă ulterioară începe cu c iar coada este o urmă a lui P . De asemenea coada trebuie să poată fi descrisă de orice specificație a lui P

L4B Dacă $P \text{ sat } S(ur)$
atunci $(c \rightarrow P) \text{ sat } (ur = \diamond \vee (ur_0 = c \wedge S(ur')))$

Un corolar al acestei legi tratează prefixul dublu

L4C Dacă $P \text{ sat } S(ur)$
atunci $(c \rightarrow d \rightarrow P) \text{ sat } (ur \leq \langle c, d \rangle \vee (ur \geq \langle c, d \rangle \wedge S(ur'')))$

Alegerea binară este similară prefixului, dar urma poate începe cu oricare din cele două evenimente alternative, iar coada este descrisă de specificația alternativei alese

Alegerea binară este similară prefixului, dar urma poate începe cu oricare din cele două evenimente alternative, iar coada este descrisă de specificația alternativei alese

L4D Dacă $P \text{ sat } S(ur)$
 și $Q \text{ sat } T(ur)$
 atunci $(c \rightarrow P | d \rightarrow Q) \text{ sat } (ur = \Diamond \vee (ur_0 = c \wedge S(ur')) \vee (ur_0 = d \wedge T(ur')))$

Toate legile descrise mai sus sunt cazuri particulare ale legii alegerii generale

L4 Dacă $\forall x \in B. (P \text{ sat } S(ur, x))$
 atunci $(x: B \rightarrow P(x)) \text{ sat } (ur = \Diamond \vee (ur_0 \in B \wedge S(ur', ur_0)))$

Legea pentru operatorul *după* este surprinzător de simplă. Dacă ur este o urmă a lui (P/s) , $s \sim ur$ este o urmă a lui P și de aceea trebuie să poată fi descrisă de orice specificație pentru P

L5 Dacă $P \text{ sat } S(ur)$
 și $s \in \text{urme}(P)$
 atunci $(P/s) \text{ sat } S(s \sim ur)$

În fine, avem nevoie de o lege care să stabilească corectitudinea procesului definit recursiv

L6 Dacă $F(X)$ este cu gardă
 și $STOP \text{ sat } S$
 și $((X \text{ sat } S) \Rightarrow (F(X) \text{ sat } S))$
 atunci $(\mu X. F(X)) \text{ sat } S$

Prin inducție putem deduce că

$$F^n(STOP) \text{ sat } S$$

Deoarece F este cu gardă, $F^n(STOP)$ descrie complet cel puțin primii n pași din comportarea lui $\mu X. F(X)$. Astfel, orice urmă a lui $\mu X. F(X)$ este o urmă pentru $F^n(STOP)$, oricare ar fi n . Această urmă trebuie, de asemenea, să satisfacă aceeași specificație S ca și $F^n(STOP)$. O justificare mai riguroasă va fi dată în paragraful 2.8 cu ocazia demonstrației matematice de la sfârșit.

Exemplu

X1 Vrem să dovedim că (1.1.2 X2, 1.10 X3)

ATS sat *SPECAL*

Demonstrație. (1) *STOP* sat ($ur = \langle \rangle$) L4A
 $\Rightarrow 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$
 deoarece $(\langle \rangle \downarrow mon) - (\langle \rangle \downarrow choc) = 0$

Concluzia rezultă folosind L3

(2) Presupunem *A* sat $(0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1)$

Deci $(mon \rightarrow choc \rightarrow \lambda)$ sat $(ur \leq \langle mon, choc \rangle$
 $\vee (ur \geq \langle mon, choc \rangle$
 $\wedge 0 \leq (ur'' \downarrow mon) - (ur'' \downarrow choc) \leq 1)$ L4C
 $\Rightarrow 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$

deoarece $\langle \rangle \downarrow mon = \langle \rangle \downarrow choc = \langle mon \rangle \downarrow choc = 0$

și $\langle mon \rangle \downarrow mon = (\langle mon, choc \rangle \downarrow mon) = \langle mon, choc \rangle \downarrow choc = 1$

și $ur \geq \langle mon, choc \rangle \Rightarrow (ur \downarrow mon = ur'' \downarrow mon + 1 \wedge ur \downarrow choc = ur'' \downarrow choc + 1)$

Concluzia rezultă din L3 și L6 □

Faptul că un proces satisface specificațiile nu înseamnă automat că este util necesităților cerute. De exemplu, avem

$$ur = \langle \rangle \Rightarrow 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$$

și putem deduce din L3 și L4A că

$$STOP \text{ sat } 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$$

Cu toate acestea *STOP* nu poate fi folosit ca un automat de vânzare, nici pentru patron, nici pentru client. Este clar că evită să facă ceva greșit, dar asta nu înseamnă nimic. Putem spune că *STOP* satisface orice specificație satisfăcătoare oricărui proces.

Din fericire, este evident că *ATS* nu se poate opri vreodată. De fapt, orice proces definit în termenii prefixului, alegerii și recursivității cu gardă nu se va opri niciodată. Singura modalitate de a descrie un proces cu posibilitatea de a se opri este de a include explicit în descrierea sa *STOP* sau $(x: B \rightarrow P(x))$, unde *B* este mulțime vidă. Prin evitarea acestor cazuri putem scrie procese care garantat nu se vor opri. Totuși, după introducerea concurenței, în următorul

capitol, vom vedea că aceste singure premise nu sunt suficiente. Modalitatea generală de a demonstra că un proces nu se oprește este dată în paragraful 3.7.

2 Concurență

2.1 Introducere

Un proces este definit prin descrierea întregii sale comportări posibile. De multe ori, există o alegere între diferitele acțiuni, de exemplu introducerea unei monezi de doi penny sau a uneia de un penny într-un ATC (1.1.3 X4). În fiecare caz, alegerea primului eveniment poate fi controlată de mediul în care evoluează procesul. De exemplu, cumpărătorul poate selecta ce monedă introduce în ATC . Din fericire, mediul unui proces poate fi descris de asemenea ca un proces cu comportarea definită prin notațiile introduse. Aceasta permite investigarea comportării unui sistem complet format din proces și mediul său, cei doi interacționând concurent. Sistemul poate fi privit ca un proces a cărui comportare este definită în termenii comportărilor asigurate pentru cele două procese componente. Sistemul poate fi plasat la rândul lui într-un mediu mai cuprinzător ș.a.m.d. De fapt, este bine de înțeles că putem șterge diferența între procese, medii și sisteme. Toate sunt procese a căror comportare poate fi imaginată, descrisă, înregistrată și analizată într-o manieră simplă și omogenă.

2.2 Interacțiune

Când două procese sunt puse să evolueze concurent, intenția este ca ele să interacționeze. Aceste interacțiuni pot fi privite ca evenimente ce necesită participarea simultană a celor două procese. Pentru cele ce urmează, să ne îndreptăm atenția spre astfel de evenimente și să le ignorăm pe celelalte. Vom presupune totodată că alfabetele celor două procese sunt identice. De asemenea, fiecare eveniment care apare trebuie să fie un eveniment posibil în comportarea independentă a fiecărui proces separat. De exemplu, o ciocolată poate fi extrasă dintr-un AT numai când cumpărătorul dorește și AT este pregătit să o facă. Fie P și Q două procese cu același alfabet, atunci introducem

$$P||Q$$

pentru a desemna procesul care se comportă ca sistemul compus din procesele P și Q ce interacționează sincronizându-se în anumite momente, conform celor expuse mai sus.

Exemple

X1 Un client pofticios este fericit de a obține o bomboană sau o ciocolată fără a plăti. Totuși, el va plăti, fără tragere de inimă, o monedă, după care insistă să primească ciocolată

$$CLPOF = (bonbon \rightarrow CLPOF \mid choc \rightarrow CLPOF \mid mon \rightarrow choc \rightarrow CLPOF)$$

Când acest consumator se găsește față în față cu $AVCB$ (1.1.3 X3) este frustrat, deoarece automatul nu permite extragerea produselor înaintea plății. Pe de altă parte, $AVCB$ nu va da nici o bomboană, pentru că consumatorul a plătit o ciocolată

$$(CLPOF||AVCB) = \mu X. (mon \rightarrow choc \rightarrow X)$$

Acest exemplu arată cum un proces definit ca o compoziție a două subprocese poate fi rescris ca un proces fără operatorul $||$. □

X2 Un consumator zăpăcit dorește o napolitană mare astfel că pune o monedă în AVC . Nu observă dacă a introdus un penny sau doi penny

$$ZĂPĂCIT = (in2p \rightarrow mare \rightarrow ZĂPĂCIT \mid in1p \rightarrow mare \rightarrow ZĂPĂCIT)$$

Din nefericire, AVC nu poate elibera o napolitană mare pentru un penny

$$(ZĂPĂCIT||AVC) = \mu X. (in2p \rightarrow mare \rightarrow X \mid in1p \rightarrow STOP)$$

$STOP$ -ul care intervine după primul eveniment $in1p$ se mai numește *blocaj*. De fapt, este situația când fiecare proces component este pregătit să se angajeze în acțiuni ulterioare diferite. Deoarece nu se pot înțelege care acțiune să fie aceasta se ajunge la blocaj. □

Exemplele anterioare pun în evidență abdicarea de la abstractizare și obiectivitate. Este important de rememorat că evenimentele sunt niște tranziții neutre care pot fi observate și înregistrate de oricine, fără a cunoaște, de exemplu, a priori, cum este o napolitană mare, mică, o ciocolată sau bomboană. Special s-a ales alfabetul format din evenimente relevante pentru a exclude

stări emoționale sau alte aspecte. Dacă dorim putem introduce evenimente pentru modelarea schimbării stărilor interne, ca în 2.3 X1.

2.2.1 Legi

Legile guvernând comportarea lui $(P||Q)$ sunt simple și regulate. Prima exprimă simetria între proces și mediu

$$\text{L1 } P||Q = Q||P$$

Următoarea lege arată că nu contează ordinea în care interacționează procesele

$$\text{L2 } P||(Q||R) = (P||Q)||R$$

Interacțiunea cu procesul aflat în blocaj duce la blocaj. O interacțiune cu procesul $RUN_{\alpha P}$ (1.1.3 X8) nu presupune vreo restricție ulterioară pentru P

$$\text{L3A } P||STOP_{\alpha P} = STOP_{\alpha P}$$

$$\text{L3B } P||RUN_{\alpha P} = P$$

Următoarele legi arată cum o pereche de procese se angajează simultan în aceeași acțiune sau se blochează dacă prima acțiune nu este comună

$$\text{L4A } (c \rightarrow P) || (c \rightarrow Q) = (c \rightarrow (P \mid Q))$$

$$\text{L4B } (c \rightarrow P) || (d \rightarrow Q) = STOP \quad \text{dacă } c \neq d$$

Aceste legi se generalizează la cazurile când unul sau ambele procese oferă o alegere pentru evenimentul inițial. Numai evenimentele pe care le oferă amândouă procesele (comune) sunt posibile de a fi angajate când procesele se combină interacționând

$$\text{L4 } (x:A \rightarrow P(x)) || (y:B \rightarrow Q(y)) = (z:(A \cup B) \rightarrow P(z) || Q(z))$$

Acastă lege permite unui sistem definit în termeni de concurență să fie transformat într-o descriere alternativă fără concurență

Exemplu

$$\text{X1 Fie } P = (a \rightarrow h \rightarrow P | b \rightarrow P)$$

$$\text{și } Q = (a \rightarrow (b \rightarrow Q | c \rightarrow Q))$$

$$\text{Atunci } (P||Q) = a \rightarrow ((b \rightarrow P) || (b \rightarrow Q | c \rightarrow Q)) \\ = a \rightarrow (b \rightarrow (P||Q))$$

din L4A

din L4A

$=\mu X.(a \rightarrow b \rightarrow X)$ deoarece recursivitatea este cu gardă \square

2.2.2 Implementare

Implementarea operatorului \parallel se bazează pe L4

$intersect(P, Q) = \lambda z. \text{if } P(z) = "BLIP" \vee Q(z) = "BLIP" \text{ then } "BLIP"$
 $\text{else } intersect(P(z), Q(z))$

2.2.3 Urme

Deoarece fiecare acțiune a lui $(P \parallel Q)$ necesită participarea simultană atât a lui P cât și a lui Q , fiecare secvență de astfel de acțiuni trebuie să fie posibilă pentru ambii operanzi. Pentru același motiv $/s$ este distributivă față de \parallel .

L1 $urme(P \parallel Q) = urme(P) \cap urme(Q)$

L2 $(P \parallel Q)/s = (P/s) \parallel (Q/s)$

2.3 Concurență

Operatorul descris în paragraful anterior poate fi generalizat la cazul când operanzii P și Q au alfabet diferite

$$\alpha P \neq \alpha Q$$

Când procesele evoluează concurrent, evenimentele care sunt în ambele alfabet (așa cum s-a arătat mai sus) necesită participarea simultană a lui P și Q . Evenimentele care aparțin exclusiv lui P nu-l interesează pe Q , care nu are cum să le observe. Astfel de evenimente pot apărea independent de Q când P se angajează în ele. Similar, Q se poate angaja în evenimente exclusiv ale lui. De aceea, mulțimea evenimentelor care sunt logic posibile pentru întregul sistem este desigur reuniunea alfabetelor proceselor componente

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

Acest operator de compunere este un exemplu mai rar de operator cu operanzi cu alfabet diferite iar rezultatul are un al 3-lea alfabet. Totuși, când cei doi operanzi au același alfabet, rezultatul are și el același alfabet și $(P \parallel Q)$ are evident semnificația anterioară

Exemple

X1 Fie $\alpha AVSUNET = \{mon, choc, bing, bang, bonbon\}$
unde *bing* este sunetul unei monezi căzând în cutia de bani a automatului de vânzare pentru care se iau în considerare și sunete
și *bang* este sunetul făcut de automat la terminarea unei operații

Automatul nu eliberează bomboane (s-au terminat)

$$AVSUNET = (mon \rightarrow bing \rightarrow choc \rightarrow bang \rightarrow AVSUNET)$$

Dacă consumatorul preferă bomboane n-are decât să blesteme, dar trebuie să ia numai ciocolate

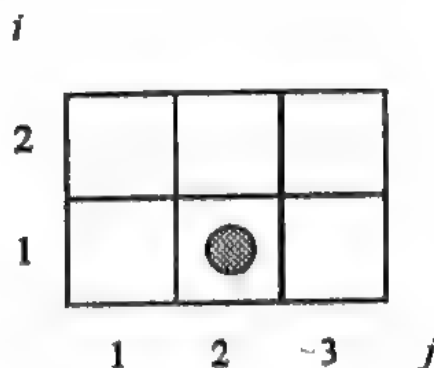
$$\begin{aligned} \alpha CONS &= \{mon, choc, blestem, bonbon\} \\ CONS &= (mon \rightarrow (bonbon \rightarrow CONS | blestem \rightarrow choc \rightarrow CONS)) \end{aligned}$$

Rezultatul activității concurente a celor două procese este

$$(AVSUNET || CONS) = \mu X. (mon \rightarrow (bing \rightarrow blestem \rightarrow choc \rightarrow bang \rightarrow X | blestem \rightarrow bing \rightarrow choc \rightarrow bang \rightarrow X))$$

Trebuie observat că *bing* poate fi intervertit cu *blestem*. Aceste evenimente pot apare în succesiune sau simultan, caz în care nu contează ordinea în care se înregistrează. De notat că formula matematică nu reprezintă în nici un fel faptul că clientul preferă o bomboană în loc de a exprima un blestem. Formula este o abstractizare a realității care ignoră emoțiile umane și se concentrează numai pe posibilitățile de apariție sau nu a evenimentelor din alfabet și nu a evenimentelor dorite sau nu, posibile sau nu. \square

X2



Un marker pornește din poziția arătată pe o suprafață riglată și se poate mișca *sus*, *jos*, *dreapta*, *stânga*

Fie $\alpha P = \{sus, jos\}$

$$P = (sus \rightarrow jos \rightarrow P)$$

$\alpha Q = \{stânga, dreapta\}$

$$Q = (dreapta \rightarrow stânga \rightarrow Q \mid stânga \rightarrow dreapta \rightarrow Q)$$

Comportarea markerului poate fi definită de

$$P \parallel Q$$

În acest exemplu, alfabetele αP și αQ nu au vreun element în comun. De asemenea, mișcările markerului sunt determinate de întreprinderea acțiunilor din procesul P cu cele din Q . Astfel de întreprinderi ale acțiunilor sunt foarte greu de descris fără concurență. De exemplu, fie R_{ij} procesul care semnifică comportarea markerului (X2) când se găsește în linia i și coloana j de pe suprafață, $i \in \{1, 2\}, j \in \{1, 2, 3\}$.

Atunci $(P \parallel Q) = R_{12}$, unde

$$R_2 = (jos \rightarrow R_{11} \mid dreapta \rightarrow R_{22})$$

$$R_{11} = (sus \rightarrow R_{21} \mid dreapta \rightarrow R_{12})$$

$$R_{22} = (jos \rightarrow R_{12} \mid stânga \rightarrow R_{21} \mid dreapta \rightarrow R_{23})$$

$$R_{12} = (sus \rightarrow R_{22} \mid stânga \rightarrow R_{11} \mid dreapta \rightarrow R_{13})$$

$$R_{23} = (jos \rightarrow R_{13} \mid stânga \rightarrow R_{22})$$

$$R_{13} = (sus \rightarrow R_{23} \mid stânga \rightarrow R_{12})$$

□

2.3.1 Legi

Primele trei legi pentru forma extinsă de concurență sunt aceleași cu legile pentru interacțiune (paragraful 2.2.1)

L1,2 \parallel este simetric și asociativ.

L3A $P \parallel STOP_{\alpha P} = STOP_{\alpha P}$

L3B $P \parallel RUN_{\alpha P} = P$

Fie $a \in (\alpha P - \alpha Q)$, $b \in (\alpha Q - \alpha P)$ și $\{c, d\} \subseteq (\alpha Q \cap \alpha P)$. Următoarele legi arată cum P se angajează singur în a , Q singur în b , iar c și d necesită participarea simultană a lui P și Q

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP$

dacă $c \neq d$

L5A $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$

L5B $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$

L6 $(a \rightarrow P) \parallel (b \rightarrow Q) = a \rightarrow (P \parallel (b \rightarrow Q)) \mid b \rightarrow ((a \rightarrow P) \parallel Q)$

Aceste legi pot fi generalizate pentru operatorul alegere generală

L7 Fie $P = (x:A \rightarrow P(x))$

și $Q = (y:B \rightarrow Q(y))$

Arunci $(P \parallel Q) = (z:C \rightarrow P' \parallel Q')$

unde $C = (A \cap B) \cup (A - \alpha Q) \cup (B - \alpha P)$

și $P' = P(z)$

$= P$

dacă $z \in A$

altfel

$Q' = Q(z)$

dacă $z \in B$

$= Q$

altfel

Legile de mai sus permit unui proces definit prin concurență să fie redefinit fără a folosi acest operator, ca în următoarele exemple

Exemple

X1 Fie $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$

Fie $P = (a \rightarrow c \rightarrow P)$

Fie $Q = (c \rightarrow b \rightarrow Q)$

De aceea $P \parallel Q = (a \rightarrow c \rightarrow P)(c \rightarrow b \rightarrow Q)$

$= a \rightarrow ((c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q))$

$= a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q))$

din definiție

din L5

din L4...(1)

De asemenea $P \parallel (b \rightarrow Q) = (a \rightarrow (c \rightarrow P) \parallel (b \rightarrow Q))$

$\mid b \rightarrow (P \parallel Q)$

din L6

$= (a \rightarrow b \rightarrow (c \rightarrow P) \parallel Q)$

$\mid b \rightarrow (P \parallel Q)$

din L5

$= (a \rightarrow b \rightarrow c \rightarrow (P \parallel (b \rightarrow Q)))$

din L4

$\mid b \rightarrow a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q)))$

(1) de mai sus

$= \mu X. (a \rightarrow b \rightarrow c \rightarrow X$

deoarece este

$\mid b \rightarrow a \rightarrow c \rightarrow X)$

cu gardă

De aceea $(P \parallel Q) = a \rightarrow c \rightarrow \mu X. (a \rightarrow b \rightarrow c \rightarrow X$

$\mid b \rightarrow a \rightarrow c \rightarrow X))$

din (1) de mai sus

□

2.3.2 Implementare

Implementarea operatorului \parallel este derivată direct din legea L7. Alfabetele operanzilor sunt reprezentate de liste finite de simboluri, A și B . Testul de apartenență folosește funcția $estemembru(x, A)$ definită în paragraful 1.7.

$(P \parallel Q)$ este implementat prin apelul funcției
 $concurent(P, \alpha P, \alpha Q, Q)$

definită astfel

$$concurent(P, A, B, Q) = aux(P, Q)$$

unde $aux(P, Q) =$

$\lambda x. \text{if } P = "BLIP \text{ sau } Q = "BLIP \text{ then } "BLIP$
 $\text{else if } estemembru(x, A) \wedge estemembru(x, B) \text{ then } aux(P(x), Q(x))$
 $\text{else if } estemembru(x, A) \text{ then } aux(P(x), Q)$
 $\text{else if } estemembru(x, B) \text{ then } aux(P, Q(x))$
 $\text{else } "BLIP$

2.3.3 Urme

Fie t urma lui $(P \parallel Q)$. Orice eveniment din t care aparține alfabetului lui P este un eveniment în evoluția lui P . Orice eveniment din t care nu aparține lui αP a apărut fără participarea lui P . Astfel $(t \upharpoonright \alpha P)$ este o urmă formată din evenimente în care a participat efectiv P și de aceea o urmă a lui P . Analog, $(t \upharpoonright \alpha Q)$ este o urmă a lui Q . Mai mult, fiecare eveniment din t trebuie să fie ori în αP ori în αQ . Acest raționament sugerează legea

$$L1 \quad urme(P \parallel Q) = \{t \mid (t \upharpoonright \alpha P) \in urme(P) \wedge (t \upharpoonright \alpha Q) \in urme(Q) \wedge t \in (\alpha P \cup \alpha Q)^*\}$$

Următoarea lege arată cum operatorul $/s$ se distribuie în compunerea paralelă

$$L2 \quad (P \parallel Q) / s = (P / (s \upharpoonright \alpha P)) \parallel (Q / (s \upharpoonright \alpha Q))$$

Când $\alpha P = \alpha Q$ avem

$$s \upharpoonright \alpha P = s \upharpoonright \alpha Q = s$$

și aceste legi sunt identice cu cele din paragraful 2.2.3.

Exemple

X1 Vezi 2.3 X1.

Fie $t1 = \langle mon, bing, blestem \rangle$

atunci $t1 \upharpoonright \alpha AVSUNET = \langle mon, bing \rangle$

este în $urme(AVSUNET)$

și $t1 \upharpoonright \alpha CONS = \langle mon, blestem \rangle$

este în $urme(CONS)$

De aceea $t1 \in urme(AVSUNET \parallel CONS)$

Un raționament similar ne arată că

$$\langle mon, blestem, bing \rangle \in urme(AVSUNET \parallel CONS)$$

Cele de mai sus confirmă că evenimentele *blestem* și *bing* pot fi intervertite în orice ordine. Ele pot să apară chiar simultan dar atunci nu le putem înregistra decât tot într-o succesiune. \square

În concluzie, urma lui $(P \parallel Q)$ este o întrețesere a urmei lui P cu urma lui Q , evenimentele comune aparând numai o dată. Dacă $\alpha P \cap \alpha Q = \{\}$, urmele sunt întrețeseri efective (paragraful 1.9.3.), conform cu paragraful 2.3 X2. La cealaltă extremă, când $\alpha P = \alpha Q$ fiecare eveniment aparține ambelor alfabetelor și sensul lui $(P \parallel Q)$ este exact cel definit pentru interacțiune (paragraful 2.2).

L3A Dacă $\alpha P \cap \alpha Q = \{\}$

$$urme(P \parallel Q) = \{s \mid \exists t. urme(P). \exists u. urme(Q). s \text{ întrețese } (t, u)\}$$

L3B Dacă $\alpha P = \alpha Q$

$$urme(P \parallel Q) = urme(P) \cap urme(Q)$$

2.4 Reprezentări grafice

Procesul P cu alfabetul $\{a, b, c\}$ este figurat ca un dreptunghi etichetat P bordat de un număr de linii, fiecare etichetată cu un eveniment diferit din alfabetul său (fig. 2.1). Analog Q , cu alfabetul $\{b, c, d\}$ este reprezentat ca în fig. 2.2.

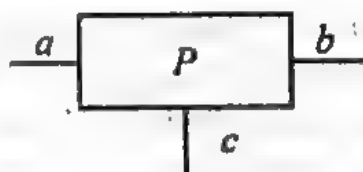


Figura 2.1

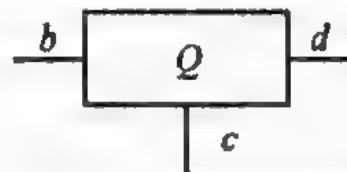


Figura 2.2

Când aceste două procese sunt puse să evolueze concurrent, sistemul rezultat poate fi desenat ca o rețea în care liniile etichetate la fel sunt conectate iar liniile etichetate cu evenimente proprii proceselor rămân libere (fig. 2.3.). Un al 3-lea proces R , unde $\alpha R = \{c, e\}$, poate fi adăugat ca în fig. 2.4.

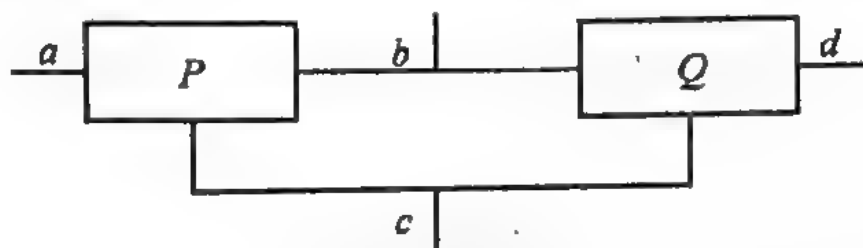


Figura 2.3

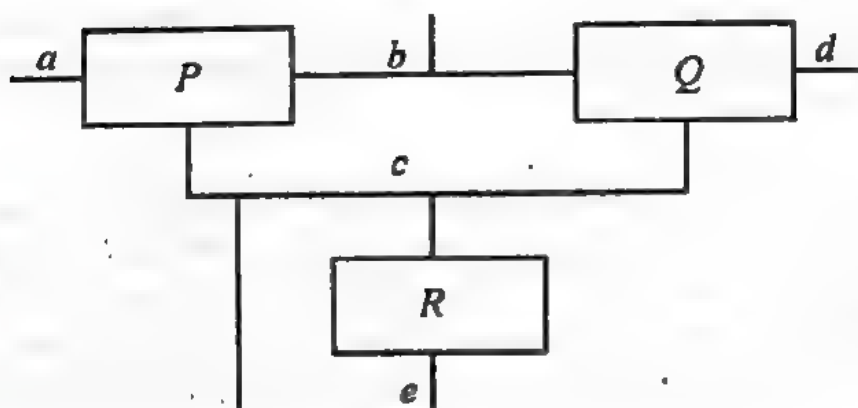


Figura 2.4

Se observă că evenimentul c necesită participarea tuturor celor trei procese, b necesită participarea lui P și Q , în timp ce evenimentele proprii a, d, e privesc un singur proces. Reprezentările de acest fel vor fi numite diagrame de conexiune.

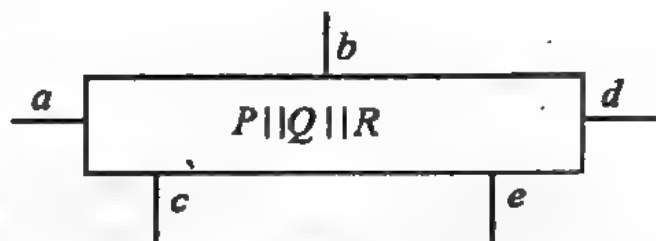


Figura 2.5

Trebuie reamintit că un sistem construit din 3 procese este totuși un singur proces și poate fi reprezentat ca un singur dreptunghi (fig. 2.5). Astfel, numărul 60 poate fi privit ca produsul a trei numere $3 \times 4 \times 5$, dar după ce l-am obținut rămâne un singur număr și maniera de construcție nu mai este relevantă sau observabilă.

2.5 Exemplu: Problema mesei celor cinci filozofi

În antichitate, un filantrop bogat a donat unei Academii cele necesare pentru a găzdui 5 filozofi eminenți. Fiecare filozof avea o cameră în care își putea desfășura activitatea sa de gândire. Exista de asemenea o sală de mese comună, cu o masă rotundă și cinci scaune etichetate cu numele filozofului ce urma să se așeze pe el. Numele filozofilor erau $FIL_0, FIL_1, FIL_2, FIL_3, FIL_4$, și ei erau dispuși în sens trigonometric în jurul mesei. În stânga fiecărui filozof era așezată o furculiță iar în centrul mesei era un castron mare cu spaghetti, ce era continuu umplut.

Un filozof își petrecea cea mai mare parte a timpului gândind. Când i se făcea foame, se ducea în sala de mese, se așeza pe propriul său scaun, își lua propria furculiță din stânga și o înfigea în castronul cu spaghetti. Dar natura încălцитă a spaghettiilor face să fie necesară încă o furculiță. De aceea filozoful avea nevoie și de furculița din dreapta. Când filozofii ce mâncau terminau, puneau jos ambele furculițe, se ridicau și se duceau în camerele lor pentru a reflecta din nou. Desigur, o furculiță putea fi folosită numai de un filozof la un moment dat. Dacă alt filozof ar fi dorit-o, el trebuia să aștepte până când devenea disponibilă.

2.5.1 Alfabet

Vom construi acum un model matematic al acestui sistem. Pentru început trebuie să selectăm mulțimea de evenimente relevante. Pentru FIL_i , mulțimea este definită astfel

$$\alpha FIL_i = \{i.așează, i.ridică, \\ i.preia_furc.i, i.preia_furc.(i \oplus 1), \\ i.elib_furc.i, i.elib_furc.(i \oplus 1)\}$$

unde \oplus este suma modulo 5, astfel că $i \oplus 1$ identifică vecinul din dreapta al filozofului i .

Se observă că alfabetele filozofilor sunt disjuncte. Nu există nici un eveniment în care să participe împreună, astfel că nu există nici o modalitate de a interacționa sau comunica între ei - o comportare reflectată realist pentru atitudinea filozofilor din acele vremuri.

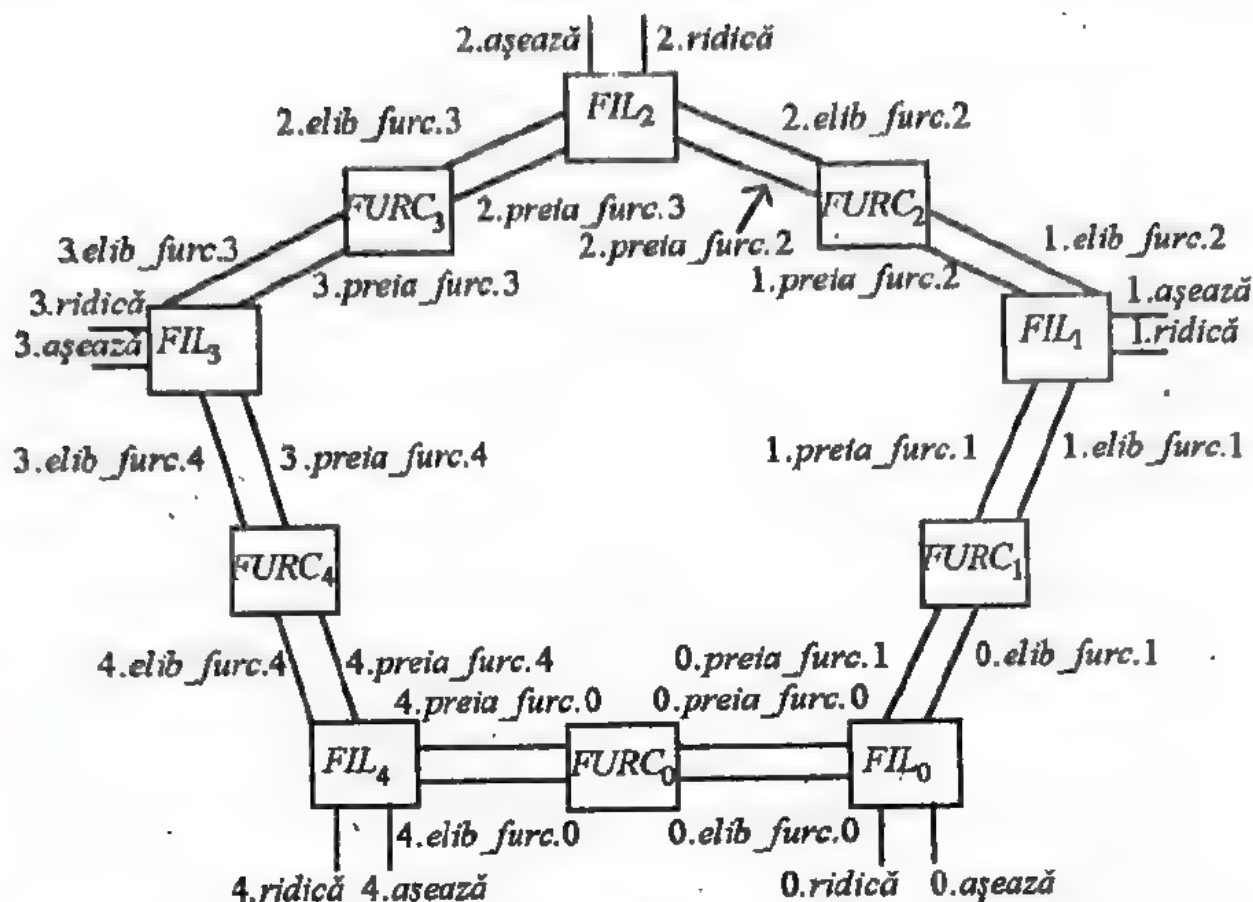


Figura 2.6

Ceilalți actori în mica noastră dramă sunt cele cinci furculițe, fiecare purtând același număr ca și filozoful în stânga căruia se află. O furculiță este preluată sau pusă înapoi pe masă de acest filozof sau de vecinul lui. Alfabetul furculiței i este definit astfel

$$\alpha FURC_i = \{i.preia_furc.i, (i \ominus 1).preia_furc.i, \\ i.elib_furc.i, (i \ominus 1).elib_furc.i\}$$

unde \ominus înseamnă scădere modulo 5.

Astfel, fiecare eveniment exceptând ridicatul sau așezatul pe scaun, necesită participarea exact a doi actori vecini, un filozof și o furculiță, așa cum se arată în fig. 2.6.

2.5.2 Comportarea

În afară de starea de a reflecta și a mânca, pe care le ignorăm, existența fiecărui filozof este descrisă de repetarea unui ciclu de șase evenimente

$$FIL_i = (i.așează \rightarrow i.preia_furc.i \rightarrow i.preia_furc.(i \oplus 1) \rightarrow \\ i.elib_furc.i \rightarrow i.elib_furc.(i \oplus 1) \rightarrow i.ridică \rightarrow FIL_i)$$

Rolul unei furculițe este simplu, ea fiind mereu ridicată și pusă jos de unul din filozofii alăturați (aceeași în ambele situații)

$$FURC_i = (i.preia_furc.i \rightarrow i.elib_furc.i \rightarrow FURC_i \\ | (i \oplus 1).preia_furc.i \rightarrow (i \oplus 1).elib_furc.i \rightarrow FURC_i)$$

Descrierea activităților din cadrul Academiei este compunerea concurentă a comportărilor elementelor sale

$$FIL_5 = (FIL_0 || FIL_1 || FIL_2 || FIL_3 || FIL_4) \\ FURC_5 = (FURC_0 || FURC_1 || FURC_2 || FURC_3 || FURC_4) \\ ACADEMIE = (FIL_5 || FURC_5)$$

O variantă interesantă a acestei istorii permite filozofilor să-și ridice furculițele în orice ordine vor sau să le așeze în orice ordine. Să considerăm comportarea mâinilor fiecărui filozof. Fiecare mână este capabilă să ridice furculița corespunzătoare, dar ambele mâini (ale aceluiași filozof) sunt necesare în situațiile de așezare sau ridicare de pe scaun

$$\alpha ST\hat{A}NGA_i = \{i.a\acute{s}ează, i.ridică, i.preia_furc.i, i.elib_furc.i\} \\ \alpha DREAPTA_i = \{i.a\acute{s}ează, i.ridică, i.preia_furc.(i \oplus 1), i.elib_furc.(i \oplus 1)\} \\ ST\hat{A}NGA_i = (i.a\acute{s}ează \rightarrow i.preia_furc.i \rightarrow i.elib_furc.i \rightarrow i.ridică \rightarrow ST\hat{A}NGA_i) \\ DREAPTA_i = (i.a\acute{s}ează \rightarrow i.preia_furc.(i \oplus 1) \rightarrow i.elib_furc.(i \oplus 1) \rightarrow i.ridică \\ \rightarrow DREAPTA_i) \\ FIL_i = ST\hat{A}NGA_i || DREAPTA_i$$

Sincronizarea pe evenimentele *așează* și *ridică* de pe scaun între $ST\hat{A}NGA_i$ și $DREAPTA_i$ ne asigură că nici o furculiță nu va fi ridicată decât când filozoful corespunzător este așezat. Astfel operațiile cu cele două furculițe adiacente unui filozof sunt întreprinse arbitrar.

În altă variantă a acestei istorii fiecare furculiță poate fi ridicată și așezată de mai multe ori în timp ce un filozof stă la masă. De aceea comportarea mâinilor este modificată pentru a conține o buclă, de exemplu

$$ST\hat{A}NGA_i = (i.a\acute{s}ează \rightarrow \mu X. (i.preia_furc.i \rightarrow i.elib_furc.i \rightarrow X \\ | i.ridică \rightarrow ST\hat{A}NGA_i))$$

2.5.3 Blocaj

Când s-a construit modelul matematic, a apărut următoarea primejdie. Să presupunem că toți filozofii devin flămânzi cam în același timp. Ei se vor așeza și ridica propria furculiță (stânga). Toți vor căuta să ridice și cealaltă furculiță

care nu e disponibilă. În această situație toți vor flămânzi și mai tare, inevitabil. Cu toate că orice proces este capabil de o acțiune ulterioară, evident nu există nici una asupra căreia două procese să se înțeleagă să o realizeze.

Totuși, istoria nu se termină așa trist. Odată pericolul detectat s-au găsit căi pentru evitarea lui. De exemplu, unul dintre filozofi ar putea ridica întâi furculița din dreapta lui sau s-ar putea achiziționa o furculiță (cinci furculițe în plus ar fi prea scump).

Soluția finală adoptată a fost numirea unui valet a cărui sarcină era să supravegheze (să conducă) așezarea și ridicarea fiecărui filozof de pe scaunul său. Alfabetul său este

$$\bigcup_{i=0}^4 \{i. așează, i. ridică\}$$

Acestui valet i s-au dat instrucțiuni secrete de a nu permite să se așeze mai mult de patru filozofi odată la masă. Comportarea lui este cel mai simplu definită recursiv mutual.

$$\text{Fie } R = \bigcup_{i=0}^4 \{i. ridică\}, A = \bigcup_{i=0}^4 \{i. așează\}$$

$VALET_i$ definește comportarea valetului cu i filozofi așezați

$$\begin{aligned} VALET_0 &= (x.A \rightarrow VALET_1) \\ VALET_j &= (x.A \rightarrow VALET_{j+1} \mid y.R \rightarrow VALET_{j-1}) \\ VALET_4 &= (y.R \rightarrow VALET_3) \end{aligned} \quad j \in \{1, 2, 3\}$$

Un proces *Academie* fără blocaj este

$$ACADEMIE_NOU\check{A} = (ACADEMIE \parallel VALET_0)$$

Istoria mesei celor cinci filozofi se datorește lui E. W. Dijkstra, iar introducerea valetului lui Carel S. Scholten.

2.5.4 Demonstrarea lipsei blocajului

În procesul *ACADEMIE* riscul blocajului era mai mult ca evident, de aceea pretenția că noul proces *ACADEMIE_NOU\check{A}* este lipsit de blocaj trebuie demonstrat cu prudență. Ceea ce trebuie să arătăm poate fi formulat

$$(ACADEMIE_NOU\check{A}/s) \neq STOP$$

$$\text{pentru toate } s \in \text{urme}(ACADEMIE_NOU\check{A})$$

Demonstrația începe cu considerarea unei urme arbitrare s iar apoi arătăm că în toate cazurile există cel puțin un eveniment prin care s poate fi extinsă și totuși rămâne în $urme(ACADEMIE_NOUĂ)$. Întâi definim numărul filozofilor așezați

$$așezați(s) = \#(s \upharpoonright A) - \#(s \upharpoonright R)$$

unde R și A sunt definiți mai sus

Din cauză că (2.3.3 L1) $s \upharpoonright (R \cup A) \in urme(VALET_0)$ deducem că

$$așezați(s) \leq 4$$

Dacă am avea $așezați(s) \leq 3$, cel puțin un filozof s -ar mai putea așeza, deci nu este blocaj. În cazul $așezați(s) = 4$, considerăm numărul filozofilor care mănâncă (au ambele furculițe ridicate). Dacă numărul lor este diferit de zero atunci un filozof care a mâncat pune furculița din stânga sa jos. Dacă nici un filozof nu mănâncă, să considerăm numărul furculițelor ridicate. Dacă acesta este mai mic sau egal cu trei, unul din filozofii care sunt așezați poate ridica furculița din stânga. Dacă sunt patru furculițe ridicate, atunci filozoful din stânga locului vacant poate ridica și furculița din dreapta. Dacă sunt cinci furculițe ridicate, cel puțin unul din filozofi mănâncă (nu avem blocaj!).

Demonstrația de mai sus implică analiza numărului de cazuri descrise formal în termenii comportării exemplului nostru. Să considerăm altă metodă de demonstrare. Să programăm un calculator pentru a găsi toate cazurile de blocaj. În general, nu putem ști dacă un program a verificat toate posibilitățile de blocaj. Dar în cazul unui sistem cu un număr finit de stări ca procesul *ACADEMIE* este suficient să considerăm acele urme a căror lungime nu depășesc limita numărului de stări. În general, numărul de stări a lui $(P \parallel Q)$ nu depășește produsul numărului de stări ale lui P înmulțit cu numărul de stări ale lui Q . Deoarece fiecare filozof are 6 stări și fiecare furculiță are 3 stări, numărul total de stări pentru *ACADEMIE* nu depășește

$$6^5 \times 3^5 \approx 1,8 \text{ milioane}$$

Deoarece alfabetul procesului *VALET* este conținut în acela al procesului *ACADEMIE*, atunci procesul *ACADEMIE_NOUĂ* nu are un număr mai mare de stări. Astfel, în aproape fiecare stare sunt posibile două sau mai multe evenimente, numărul urmelor care pot fi examinate în problema noastră depășind 21,8 milioane. Este clar că nici un calculator nu va putea explora toate aceste posibilități. De aceea, demonstrarea lipsei blocajului chiar pentru procese finite simple va rămâne responsabilitatea proiectantului de sisteme concurente.

2.5.5 Preluare infinită

În afară de pericolul blocajului mai există un pericol pentru sistemul analizat – acela ca un filozof care intenționează să mănânce să fie luat prin surprindere de un vecin. Să presupunem că în stânga unuia din filozofi există un vecin destul de lacom și că primul are de asemenea o mână stângă mai nesigură (de aceea mișcarea cu ea este încetinită). Astfel, înainte ca el să-și ridice furculița lui stângă, vecinul lacom (care este și foarte îndemânatic) se grăbește, se așează, acaparează ambele furculițe și petrece un timp destul de lung mâncând. Apoi, el eliberează ambele furculițe și pleacă. Dar când flămânzește din nou, se grăbește la masă, se așează, ridică furculițele, în timp ce vecinul său nu mai apucă să ridice și el furculița comună. Deoarece acest ciclu se poate repeta la infinit, unul din filozofii așezați poate rămâne nemâncat.

Soluția corectă la această situație este să o privim ca o chestiune insolubilă, deoarece în situația că unul din filozofi se comportă ca cel lacom atunci un alt filozof va rămâne flămând o perioadă mai mare decât în mod obișnuit. Astfel se observă că nu există soluție care să aducă satisfacție totală, rezolvarea sigură fiind de a cumpăra mai multe furculițe și mai multe spaghetti.

Totuși este important de garantat că un filozof așezat va mânca, ceea ce implică modificarea comportării valetului. Ajutând un filozof să se așeze, el așteaptă până când acesta ridică ambele furculițe după care ajută un alt vecin să se așeze. Dar rămâne situația când unul din filozofi poate fi luat prin surprindere mereu. Să presupunem că valetul are o antipatie nejustificată față de unul din filozofi și permanent întârzie acțiunea de a-l conduce la scaun, chiar dacă filozoful este pregătit pentru acest eveniment. Aceasta este o altă posibilitate care nu a fost luată în considerare în exemplul nostru deoarece nu o putem distinge de situația când un filozof însuși așteaptă un timp foarte lung până flămânzește. Avem de a face cu o problemă de sincronizare pe care am hotărât să o ignorăm deliberat sau rămâne în sarcina unei alte faze a proiectării și implementării. Este sarcina celui care implementează să se asigure că orice eveniment ce devine posibil va avea loc într-un interval de timp rezonabil. De asemenea, implementarea în limbaje de nivel înalt nu trebuie să introducă asemenea întârzieri arbitrare în execuția programului, chiar dacă programatorul nu are mijloace de a impune sau descrie acest lucru.

2.6 Schimbare de simbol

Exemplul din paragraful precedent a făcut apel la două mulțimi de procese : filozofii și furculițele. În cadrul fiecărei mulțimi procesele au comportări foarte apropiate cu excepția numelor evenimentelor în care se angajează. În acest paragraf vom introduce o metodă potrivită pentru definirea unor procese cu

comportări similare. Fie f o injecție (unu la unu) ce realizează corespondența între alfabetul lui P și mulțimea de simboluri A

$$f: \alpha P \rightarrow A$$

Definim procesul $f(P)$ ca unul care se angajează în evenimentul $f(c)$ ori de câte ori P se angajează în c . Avem

$$\alpha f(P) = f(\alpha P)$$

$$\text{urme}(f(P)) = \{f^*(s) \mid s \in \text{urme}(P)\}$$

(Pentru definirea lui f^* vezi 1.9.1.).

Exemple

X1 În timp, prețurile produselor, după cum prea bine știm, cresc. Pentru a reprezenta efectul inflației, definim o funcție f prin următoarele ecuații

$$\begin{array}{ll} f(\text{in}2p) = \text{in}5p & f(\text{mare}) = \text{mare} \\ f(\text{in}1p) = \text{in}5p & f(\text{mic}) = \text{mic} \\ f(\text{ex}1p) = \text{ex}5p & \end{array}$$

Noul automat de vânzare complex este

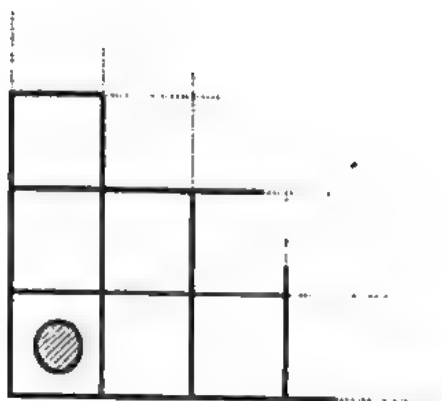
$$AVCNOU = f(AVC) \quad \square$$

X2 Un obiect se comportă ca MM_0 (1.1.4 X2), dar efectuează acțiunile, *stânga*, *dreapta* în loc de *sus*, *jos*

$$\begin{array}{l} f(\text{sus}) = \text{dreapta}, \quad f(\text{jos}) = \text{stânga}, \quad f(\text{pe_loc}) = \text{pe_loc}, \\ SD_0 = f(MM_0) \end{array} \quad \square$$

Principalul scop al tehnicii de schimbare a numelor evenimentelor este de a permite o compunere mai eficientă a proceselor în construcții concurente.

X3 Un marker se mișcă pe o suprafață riglată infinită *stânga*, *dreapta*, *sus*, *jos*, margini existând doar în partea stângă și în cea de jos



Markerul pornește din colțul din stânga jos. Ca și în 2.3 X2, mișcările pe verticală și orizontală pot fi modelate prin acțiuni independente ale unor procese separate. Numai în colțul din stânga jos markerul se poate angaja în evenimentul *pe_loc*, care însă necesită participarea simultană a celor două procese separate

$$SDSJ = SD_0 \parallel MM_0$$

□

X4 Vrem să conectăm două instanțe ale lui *COPIEBIT* (1.1.3 X7), astfel că fiecare bit emis de primul să fie primit de al doilea. Mai întâi trebuie să modificăm numele evenimentelor folosite pentru comunicația internă, așa încât vom introduce două noi evenimente *mij.0* și *mij.1* și definim funcțiile *f* și *g* pentru a schimba emiterea unuia din procese și primirea celuilalt

$$\begin{aligned} f(emi.0) &= g(pri.0) = mij.0 \\ f(emi.1) &= g(pri.1) = mij.1 \\ f(pri.0) &= pri.0, f(pri.1) = pri.1 \\ g(emi.0) &= emi.0, g(emi.1) = emi.1 \end{aligned}$$

Rezultatul pe care-l așteptăm este

$$LANȚ2 = f(COPIEBIT) \parallel g(COPIEBIT)$$

De observat că fiecare emisie a unui 0 sau 1 de către operandul stâng al operatorului \parallel este (prin definiția lui *f* și *g*) același eveniment (*mij.0*, *mij.1*) ca cel primit drept 0 sau 1 de către operandul din dreapta. Astfel este modelată comunicarea sincronizată a unor biți printr-un canal ce leagă două procese, ca în fig. 2.7



Figura 2.7

Operandul din stânga nu are de ales în ce privește valoarea emisă pe canalul de legătură, în timp ce operandul din dreapta este pregătit să se angajeze în oricare din evenimentele $mij.0$, $mij.1$. De aceea revine procesului care emite să determine cu fiecare ocazie care dintre cele două evenimente va apărea. Această metodă de comunicație între procese concurente va fi generalizată în cap. 4.

Să remarcăm că evenimentele $mij.0$ și $mij.1$ sunt în alfabetul procesului compus și pot fi observate (chiar controlate) de mediu. Cineva ar putea avea intenția de a ignora sau masca astfel de evenimente interne, dar în general mascarea ar putea duce la nedeterminism, astfel că problema este amânată pentru paragraful 3.5 \square

X5 Vrem să reprezentăm comportarea unei variabile booleene dintr-un program oarecare. Evenimentele din alfabetul său sunt

<i>assign0</i>	asignează variabilei valoarea 0
<i>assign1</i>	asignează variabilei valoarea 1
<i>acces0</i>	accesează valoarea variabilei când este 0
<i>acces1</i>	accesează valoarea variabilei când este 1

Comportarea variabilei este foarte asemănătoare cu cea a automatului de băuturi (1.1.4 X1), astfel că definim

$$BOOL = f(AK)$$

unde definirea lui f este o chestiune banală. Se observă că nu se poate accesa o valoare până ce nu s-a asignat una. Încercarea de a accesa o valoare neasignată duce la blocaj - una din erorile subtile de programare. \square

Arborescența lui $f(P)$ poate fi construită din cea a lui P aplicând simplu funcția f etichetelor tuturor ramurilor. Deoarece f este o funcție injectivă, transformarea păstrează structura de arbore și distincția între etichetele arcelor pornind din același nod. De exemplu, reprezentarea lui $AVCNOU$ este dată în fig. 2.8

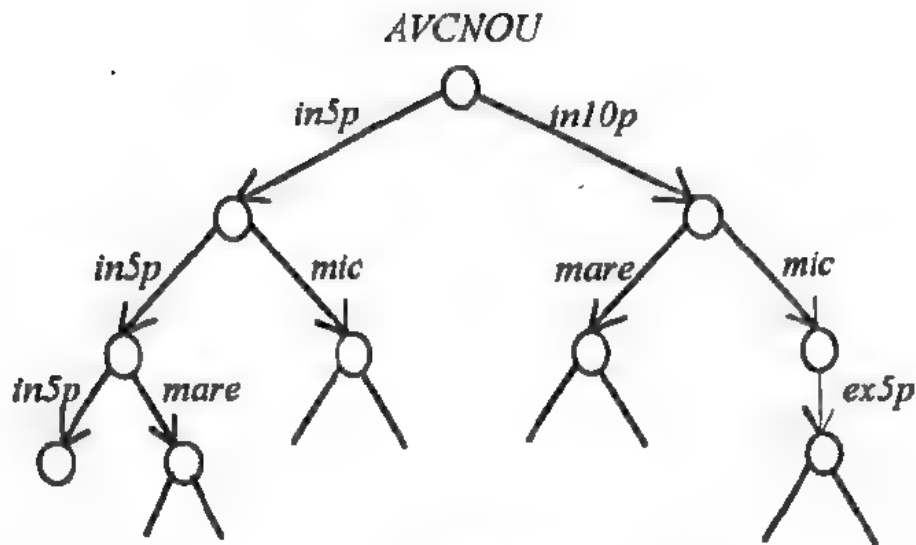


Figura 2.8

2.6.1 Legi

Schimbarea de simbol prin aplicarea unei funcții injective nu schimbă structura comportării unui proces. Aceasta rezultă din distributivitatea aplicării unei funcții față de alți operatori, așa cum rezultă din următoarele legi. Vom folosi următoarele definiții auxiliare

$$f(B) = \{f(x) \mid x \in B\}$$

f^{-1} inversa lui f

$f \circ g$ compunerea lui f cu g

f^* definită în paragraful 1.9.1

(nevoia de f^{-1} în următoarele legi este un motiv important de a insista ca f să fie injectivă)

După schimbarea de simbol, *STOP* tot nu se angajează în nici un eveniment din alfabetul său schimbat

$$L1 \quad f(STOP_A) = STOP_{f(A)}$$

În cazul unei alegeri, atât simbolurile oferite pentru selecție cât și comportarea ulterioară sunt schimbate

$$L2 \quad f(x:B \rightarrow P(x)) = (y:f(B) \rightarrow f(P(f^{-1}(y))))$$

Utilizarea lui f^{-1} din partea dreaptă ne obligă la următoarele explicații. Să ne amintim că P reprezintă o funcție-proces ce are o comportare ulterioară depinzând de un x din mulțimea B . Dar variabila y din partea dreaptă ia valori din mulțimea $f(B)$. Astfel evenimentul corespunzător pentru P este $f^{-1}(y)$, care este

în B (deoarece $y \in f(B)$) iar comportarea lui P după acest eveniment este $P(f^{-1}(y))$, acțiunile acestui proces fiind corespunzător schimbate mai departe prin aplicarea lui f .

Schimbarea de simbol se distribuie în compunerea paralelă

$$L3 \quad f(P \parallel Q) = f(P) \parallel f(Q)$$

Schimbarea de simbol se distribuie într-un mod ceva mai complex în recursivitate, alfabetul modificându-se adecvat

$$L4 \quad f(\mu X:A. F(X)) = (\mu Y: f(A). f(F(f^{-1}(Y))))$$

Din nou, utilizarea lui f^{-1} în partea dreaptă necesită unele explicații. Să ne amintim că o condiție pentru validitatea recursivității din partea stângă cere ca F să fie o funcție cu argument un proces ce are alfabetul A , generând un proces cu același alfabet. În partea dreaptă, Y este o variabilă modelând procesele cu alfabetul $f(A)$ și nu poate fi folosită ca argument al lui F , alfabetul său trebuind să fie mulțimea A . Aceasta se realizează aplicând funcția inversă f^{-1} variabilei Y . Astfel $F(f^{-1}(Y))$ are alfabetul A , astfel că aplicarea lui f va transforma alfabetul la $f(A)$, asigurând validitatea recursivității și în partea dreaptă a legii.

Compunerea a două schimbări de simbol este definită prin compunerea a două funcții schimbare de simbol

$$L5 \quad f(g(P)) = (f \circ g)(P)$$

Urmele unui proces după schimbarea de simbol se obțin simplu prin schimbarea simbolilor individuali în orice urmă a procesului original

$$L6 \quad \text{urme}(f(P)) = \{f^*(s) \mid s \in \text{urme}(P)\}$$

Explicația următoarei și ultimei legi este similară cu aceea a lui L6

$$L7 \quad f(P)/f^*(s) = f(P/s)$$

2.6.2 Etichetarea proceselor

Schimbarea de simbol este în special utilă în construirea claselor de procese similare concurente, relativ la acțiunile față de mediu, dar care nu interacționează de loc între ele. Aceasta înseamnă că procesele trebuie să aibă alfabet disjuncte mutual. Pentru a realiza aceasta, fiecare proces este etichetat diferit, fiecare eveniment al său având aceeași etichetă ca și procesul. Un

eveniment etichetat va fi o pereche $l.x$, unde l este eticheta iar x este un simbol corespunzător unui eveniment. Un proces P etichetat cu l este notat

$$l:P$$

Acest proces se angajează în evenimentul $l.x$ ori de câte ori P se angajează în x . Evident etichetarea poate fi definită cu ajutorul unei funcții

$$\begin{aligned} f_l(x) &= l.x && \text{pentru toți } x \text{ din } \alpha P \\ l:P &= f_l(P) \end{aligned}$$

Exemple

X1 O pereche de automate care stau unul lângă altul

$$(st\acute{a}nga:AVS) || (dreapta:AVS)$$

Alfabetele celor două procese sunt disjuncte și orice eveniment care apare are eticheta automatului unde se produce. Dacă automatele nu ar fi fost etichetate înainte de a fi fost așezate, acționând în paralel, orice eveniment ar fi necesitat participarea ambelor mașini. Astfel perechea nu s-ar fi deosebit de un singur automat. Aceasta este o consecință a faptului că

$$(AVS || AVS) = AVS$$

□

Etichetarea proceselor permite folosirea lor ca și variabilele declarate local unei proceduri într-un limbaj de nivel înalt.

X2 Comportarea unei variabile boolene este modelată prin procesul *VARB* (2.6 X5). Comportarea unei proceduri este descrisă de procesul *UTIL*, care asignează și încarcă valorile a două variabile boolene numite b și c . Astfel $\alpha UTIL$ include evenimente compuse ca

$b.assign0$ asignează valoarea 0 lui b
 $c.acces1$ accesează valoarea curentă a lui c când este 1.

Procesul *UTIL* interacționează în paralel cu cele două variabile boolene

$$b:VARB || c:VARB || UTIL$$

În procedura *UTIL* se pot realiza următoarele acțiuni

$$b := false; P \text{ prin } (b.assign0 \rightarrow P)$$

$$b := \neg c; P \quad \text{prin } (c.\text{acces}0 \rightarrow b.\text{assign}1 \rightarrow P \\ | c.\text{acces}1 \rightarrow b.\text{assign}0 \rightarrow P)$$

De observat cum valoarea curentă a variabilei este identificată prin alegerea între *acces0* și *acces1*. Această alternativă afectează într-un mod adecvat comportarea ulterioară a lui *UTIL*. \square

În exemplul X2 și în următoarele ar fi mult mai comod să definim efectul unei singure asignări, de exemplu

$$b := \text{false}$$

decât perechea

$$b := \text{false}; P$$

care menționează explicit restul de program din *P*. Semnificația perechii din urmă va fi introdusă în cap. 5.

X3 Un proces *UTIL* prelucrează două variabile contor etichetate cu *l* și *m*. Ele sunt inițializate cu 0 și respectiv 3. Procesul *UTIL* le incrementează folosind evenimentele *l.sus* sau *m.sus* și le decrementează (când sunt pozitive) prin *l.jos* sau *m.jos*. Un test de zero este asigurat de evenimentele *l.pe_loc* și *m.pe_loc*. Astfel, pentru variabilele *l* și *m* poate fi folosit procesul MM_n (1.1.4 X2) după o etichetare adecvată

$$(l:MM_0 || m:MM_3 || UTIL)$$

În interiorul procesului *UTIL* se realizează următoarele efecte (exprimate convențional)

$$\begin{array}{ll} (m := m + 1; P) & \text{prin } (m.\text{sus} \rightarrow P) \\ \text{if } l = 0 \text{ then } P \text{ else } Q & \text{prin } (l.\text{pe_loc} \rightarrow P | l.\text{jos} \rightarrow l.\text{sus} \rightarrow Q) \end{array}$$

Să remarcăm cum se realizează testul pentru zero : există o alegere între evenimentul *l.jos*, pentru decrementarea contorului cu unu și evenimentul *l.pe_loc*. Contorul etichetat cu *l* selectează între aceste două evenimente : dacă valoarea variabilei este zero este selectat *l.pe_loc* iar dacă este diferită de zero, cealaltă alternativă. Dar în al 2-lea caz valoarea variabilei a fost decrementată și restabilirea ei la valoarea originală se face prin *l.sus*. În următorul exemplu restabilirea valorii originale este mai complexă.

$$(m := m + 1; P)$$

este implementată prin *ADUN*

unde $ADUN$ este definit recursiv

$$\begin{aligned} ADUN &= JOS_0 \\ \text{și } JOS_i &= (l.sos \rightarrow JOS_{i+1} | l.pe_loc \rightarrow SUS_i) \\ \text{și } SUS_0 &= P \\ \text{și } SUS_{i+1} &= l.sus \rightarrow m.sus \rightarrow SUS_i \end{aligned}$$

Procesele JOS_i testează valoarea inițială a lui l decrementând-o până la zero. Procesele SUS_i adună valoarea găsită atât la m cât și la l , refăcându-l astfel pe l la valoarea inițială și adunându-l practic pe l la m . \square

Efectul unui masiv de variabile poate fi realizat printr-o colecție de procese concurente, fiecare etichetat cu un index din masiv.

X4 Scopul procesului $TEST$ este de a înregistra dacă evenimentul ap a apărut sau nu. La prima apariție a lui ap răspunde nu și la fiecare din aparițiile următoare răspunde da

$$\begin{aligned} \alpha TEST &= \{ap, nu, da\} \\ TEST &= ap \rightarrow nu \rightarrow \mu X. (ap \rightarrow da \rightarrow X) \end{aligned}$$

Acest proces poate fi folosit pentru a simula comportarea unui masiv cu elemente întregi

$$INT3 = (0:TEST) || (1:TEST) || (2:TEST) || (3:TEST)$$

Întregul masiv poate fi iarăși etichetat înainte de utilizare

$$m:INT3 || UTIL$$

Fiecare eveniment în $\alpha(m:INT3)$ este un triplet, de exemplu $m.2.ap$. În procesul $UTIL$, efectul pentru

$$\text{if } 2 \in m \text{ then } P \text{ else } (m := m \cup \{2\}; Q)$$

poate fi realizat prin

$$m.2.ap \rightarrow (m.2.da \rightarrow P | m.2.nu \rightarrow Q) \quad \square$$

2.6.3 Implementare

În general, pentru a implementa schimbarea de simbol trebuie să știm inversa g a funcției f . Trebuie de asemenea să ne asigurăm că g va da un răspuns spe-

cial "BLIP când este aplicată la un argument din afara domeniului lui f . Implementarea este bazată pe 2.6.1 L4.

$$\begin{aligned} schimb(g,P) = & \lambda x. \text{ if } g(x) = \text{"BLIP"} \text{ then "BLIP"} \\ & \text{ else if } P(g(x)) = \text{"BLIP"} \text{ then "BLIP"} \\ & \text{ else } schimb(g,P(g(x))) \end{aligned}$$

Cazul special al etichetării procesului poate fi implementat mult mai simplu. Evenimentul compus $l.x$ este reprezentat ca o pereche de atomi $cons("l",x)$. Deci $(l:P)$ este implementat de

$$\begin{aligned} etich(l,P) = & \lambda y. \text{ if } null(y) \vee atom(y) \text{ then "BLIP"} \\ & \text{ else if } car(y) \neq l \text{ then "BLIP"} \\ & \text{ else if } P(cdr(y)) = \text{"BLIP"} \text{ then "BLIP"} \\ & \text{ else } etich(l,P(cdr(y))) \end{aligned}$$

2.6.4 Etichetare multiplă

Definiția etichetării poate fi extinsă pentru a permite fiecărui eveniment să ia orice etichetă l dintr-o mulțime L . Dacă P este un proces, $(L:P)$ este definit ca un proces care se comportă exact ca P , cu excepția faptului că se angajează în evenimentul $l.c$ (unde $l \in L$ și $c \in \alpha P$) ori de câte ori P s-ar angaja în c . Alegerea unei etichete l este făcută independent, cu orice ocazie, de mediul lui $(L:P)$.

Exemple

X1 Relativ la problema mesei celor cinci filozofi, introducem un valet tânăr care își ajută numai stăpânul său înspre și de la scaun și stă în spatele acestui scaun când acesta mănâncă

$$\begin{aligned} \alpha VAL_T = & \{așează, ridică\} \\ VAL_T = & (așează \rightarrow ridică \rightarrow VAL_T) \end{aligned}$$

Acest proces poate să-și împartă serviciile la cinci stăpâni (dar servind numai pe unul la un moment dat)

$$\begin{aligned} E = & \{0,1,2,3,4\} \\ VAL_T_M = & (E:VAL_T) \end{aligned}$$

Valetul tânăr multiplu ar putea fi angajat pentru a proteja de înfometare filozofii care iau masa când valetul (2.5.3.) este în vacanță. Desigur, filozofii

ar putea deveni mai flămânzi în vacanță deoarece numai unul dintre ei este condus la masă la un moment dat. \square

Dacă E conține mai mult decât o etichetă, imaginea arborescentă a lui $E:P$ este similară cu aceea pentru P dar mai stufoasă, în sensul că sunt mult mai multe ramuri plecând din același nod. De exemplu, imaginea lui VAL_T este un trunchi fără ramuri (fig. 2.9). Totuși imaginea lui $\{0,1\}:VAL_T$ este un arbore binar complet (fig. 2.10). Arborele pentru VAL_T_M este mult mai stufoș.



Figura 2.9

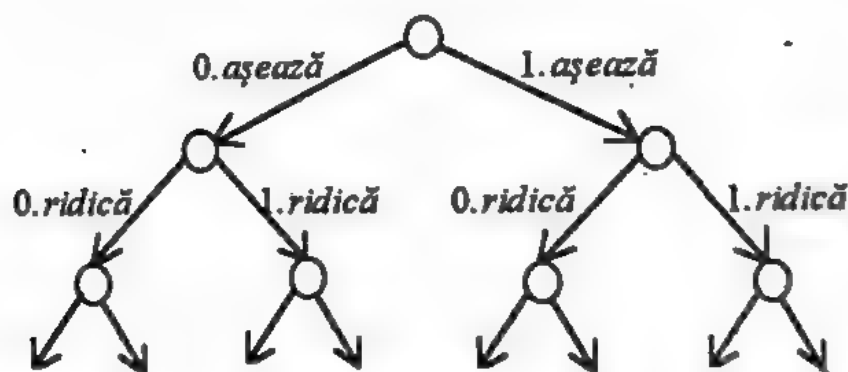


Figura 2.10

În general, etichetarea multiplă este utilă când folosim serviciile unui singur proces într-un număr de alte procese etichetate, admitând că mulțimea etichetelor este cunoscută dinainte. Această tehnică va fi exploatată mai intens în cap. 6.

2.7 Specificații

Fie P și Q procese care doresc să activeze concurrent și să presupunem că am demonstrat

$$P \text{ sat } S(ur) \\ \text{și } Q \text{ sat } T(ur)$$

Fie ur o urmă a lui $(P||Q)$. Din 2.3.3 L1 avem că $(ur \upharpoonright \alpha P)$ este o urmă a lui P și deci satisface S

$$S(ur \upharpoonright \alpha P)$$

Similar, $(ur \upharpoonright \alpha Q)$ este urmă a lui Q așa că

$$T(ur \upharpoonright \alpha Q)$$

Acest argument este valabil pentru orice urmă a lui $(P||Q)$. De aceea putem deduce

$$(P||Q) \text{ sat } (S(ur \upharpoonright \alpha P) \wedge T(ur \upharpoonright \alpha Q))$$

Acest raționament formal este rezumat în legea

$$\text{L1} \quad \text{Dacă } P \text{ sat } S(ur) \\ \text{și } Q \text{ sat } T(ur) \\ \text{atunci } (P||Q) \text{ sat } (S(ur \upharpoonright \alpha P) \wedge T(ur \upharpoonright \alpha Q))$$

Exemplu

$$\text{X1} \quad (\text{Vezi 2.3.1. X1})$$

$$\text{Fie } \alpha P = \{a, c\}, \alpha Q = \{b, c\}$$

$$\text{Fie } P = (a \rightarrow c \rightarrow P)$$

$$\text{Fie } Q = (c \rightarrow b \rightarrow Q)$$

Vrem să dovedim că

$$(P||Q) \text{ sat } 0 \leq ur \downarrow a - ur \downarrow b \leq 2$$

Demonstrația lui 1.10.2 X2 poate fi evident adaptată pentru a arăta că

$$P \text{ sat } (0 \leq ur \downarrow a - ur \downarrow c \leq 1) \\ \text{și } Q \text{ sat } (0 \leq ur \downarrow c - ur \downarrow b \leq 1)$$

Din L1 rezultă că

$$(P||Q) \text{ sat } (0 \leq (ur \upharpoonright \alpha P) \downarrow a - (ur \upharpoonright \alpha P) \downarrow c \leq 1 \wedge 0 \leq (ur \upharpoonright \alpha Q) \downarrow c - (ur \upharpoonright \alpha Q) \downarrow b \leq 1) \\ \Rightarrow 0 \leq ur \downarrow a - ur \downarrow b \leq 2 \quad \text{deoarece } (ur \upharpoonright A) \downarrow a = ur \downarrow a \text{ ori de câte ori } a \in A \quad \square$$

Deoarece legile pentru *sat* permit lui *STOP* să îndeplinească orice specificație, raționamentul bazat pe aceste legi nu poate niciodată dovedi absența blocajului, legi mai puternice urmând a fi date în paragraful 3.7. Pe moment, o cale de a elimina riscul blocării este o demonstrație atentă ca în paragraful 2.5.4. O altă metodă este de a arăta că procesul definit cu operatorul pentru concurență este echivalent cu un proces fără blocaj definit fără acest operator, așa cum a fost făcut în 2.3.1 X1. Totuși asemenea demonstrații presupun transformări lungi și anevoioase. Ori de câte ori este posibil trebuie apelat la legi generale ca

L2 Dacă *P* și *Q* nu se opresc (blochează) niciodată și dacă $(\alpha P \cap \alpha Q)$ conține cel puțin un eveniment, atunci $(P||Q)$ definit în X1 nu se va opri niciodată.

Exemplu

X2 Procesul $(P||Q)$ definit în X1 nu se va opri niciodată din cauză că

$$\alpha P \cap \alpha Q = \{c\}.$$

□

Legea pentru schimbare de simbol este

L3 Dacă *P* sat *S*(*ur*)
atunci $f(P)$ sat $S(f^{-1*}(ur))$

Utilizarea lui f^{-1*} în această lege necesită explicații. Fie *ur* o urmă a lui $f(P)$. Atunci $f^{-1*}(ur)$ este o urmă a lui *P*. Ipoteza lui L3 afirmă că orice urmă a lui *P* satisface *S*. Urmează că $f^{-1*}(ur)$ satisface *S*, ceea ce este exact concluzia lui L3.

2.8 Teoria matematică a proceselor deterministe

În descrierea proceselor am enunțat un mare număr de legi și ocazional le-am folosit în demonstrații. Legile au fost justificate (în cele mai multe din cazuri) prin raționamente elementare în raport de ce ne așteptam și doream să fie adevărat. Pentru un cititor cu reflexe de matematică aplicată sau inginerie aceasta poate fi de ajuns. Dar întrebarea care se naște este limpede : sunt aceste legi adevărate de fapt ? Sunt ele consistente ? Ar trebui să fie mai multe ? Sau sunt complete în sensul că ele permit ca toate problemele legate de

procese să fie dovedite pe baza lor ? S-ar putea cineva descurca cu mai puține legi sau mai simple ? Acestea sunt întrebări la care se poate răspunde printr-o investigație matematică mai profundă.

2.8.1 Definiții de bază

În construirea unui model matematic al unui sistem fizic o metodă bună este de a defini conceptele de bază în termenii proprietăților care pot fi direct sau indirect observate sau măsurate. Pentru un proces determinist P suntem familiarizați cu două asemenea proprietăți

- αP mulțimea evenimentelor în care procesul este evident capabil de a se angaja
- $urme(P)$ mulțimea secvențelor de evenimente în care procesul poate participa dacă este nevoie.

Am explicat cum aceste două mulțimi trebuie să satisfacă cele trei legi din 1.8.1, L6, L7, L8. Considerăm acum o pereche arbitrară de mulțimi (A, S) care satisface aceste trei legi. Perechea identifică unic un proces P a cărui urme S sunt construite potrivit următoarele definiții

Fie $P^0 = \{x \mid \langle x \rangle \in S\}$
 și $P(x)$ procesul ale cărui urme sunt $\{t \mid \langle x \rangle \wedge t \in S\}$ pentru toți x în P^0

Atunci $\alpha P = A$
 și $P = (x: P^0 \rightarrow P(x))$

Mai mult, perechea (A, S) poate fi formată din ecuațiile

$$\begin{aligned} A &= \alpha P \\ S &= urme(x: P^0 \rightarrow P(x)) \end{aligned}$$

Din această cauză există o corespondență biunivocă între fiecare proces P și perechea de mulțimi $(\alpha P, urme(P))$. În matematică aceasta este o justificare suficientă pentru echivalarea celor două concepte.

D0 Un proces determinist este o pereche

$$(A, S)$$

unde A este orice mulțime de simboluri și S este orice submulțime a lui A^* care satisface cele două condiții

$$C0 \quad \diamond \in S$$

$$C1 \quad \forall s, t. s \wedge t \in S \Rightarrow s \in S$$

Cel mai simplu exemplu de proces care îndeplinește această definiție este acela care nu face nimic

$$D1 \quad STOP_A = (A, \{\diamond\})$$

La cealaltă extremă este procesul care activează continuu

$$D2 \quad RUN_A = (A, A^*)$$

Diferenții operatori de la procese pot fi acum definiți formal arătând cum alfabetul și urmele rezultatului provin din alfabetul și urmele operanzilor.

$$D3 \quad (x:B \rightarrow (A, S(x))) = (A, \{\diamond\} \cup \{\langle x \rangle \wedge s \mid x \in B \wedge s \in S(x)\}) \text{ dacă } B \subseteq A$$

$$D4 \quad (A, S) \wedge s = (A, \{t \mid (s \wedge t) \in S\}) \text{ dacă } s \in S$$

$$D5 \quad \mu X.A.F(X) = (A, \bigcup_{n \geq 0} \text{urme}(F^n(STOP_A)))$$

$$D6 \quad (A, S) \parallel (B, T) = (A \cup B, \{s \mid s \in (A \cup B)^* \wedge (s \upharpoonright A) \in S \wedge (s \upharpoonright B) \in T\})$$

$$D7 \quad f(A, S) = (f(A), \{f^*(s) \mid s \in S\}) \text{ dacă } f \text{ este injectivă}$$

Desigur, este necesar să dovedim că în partea dreaptă a acestor definiții sunt de fapt procese, cu alte cuvinte satisfac condițiile C0 și C1 din D0. Din fericire aceasta este ușor de arătat.

În cap. 3 se va vedea că D0 nu este o definiție adecvată a conceptului de proces din cauză că nu cuprinde posibilitatea de nedeterminism. De aceea, este necesară o definiție mai generală și mai complicată. Toate legile pentru procese nedeterminate sunt adevărate pentru cele deterministe dar procesele deterministe au câteva legi în plus, de exemplu

$$P \parallel P = P$$

Pentru a evita confuzia, în această carte am evitat marcarea unor astfel de legi, astfel că toate legile pot fi folosite în siguranță la procese nedeterminate ca și la cele deterministe (cu excepția 2.2.1 L3A, 2.2.3 L1, 2.3.1 L3A, 2.3.3 L1, L2, L3, care sunt false pentru procese conținând CHAOS (3.8)).

2.8.2 Teoria de punct fix

Obiectivul acestui paragraf este de a contura demonstrația teoremei fundamentale a recursivității, astfel ca un proces definit recursiv (2.8.1 D5) să fie soluție a ecuației sale recursive

$$\mu X.F(X) = F(\mu X.F(X))$$

Metoda urmează teoria de punct fix a lui Scott. Mai întâi trebuie să stabilim o relație de ordine între procese \sqsubseteq

$$D1 \quad (A, S) \sqsubseteq (B, T) \Leftrightarrow (A = B \wedge S \subseteq T)$$

Două procese sunt comparabile în această ordine dacă au același alfabet și unul face tot ce face celălalt și chiar mai mult. Această ordonare este parțială în sensul că

$$L1 \quad P \sqsubseteq P$$

$$L2 \quad P \sqsubseteq Q \wedge Q \sqsubseteq P \Rightarrow P = Q$$

$$L3 \quad P \sqsubseteq Q \wedge Q \sqsubseteq R \Rightarrow P \sqsubseteq R$$

Un lanț în ordinea parțială este o secvență infinită de elemente

$$\{P_0, P_1, P_2, \dots\}$$

astfel că $P_i \sqsubseteq P_{i+1}$

pentru toți i

Definim limita (cea mai mică margine superioară) unui astfel de lanț

$$\bigsqcup_{i \geq 0} P_i = (\alpha P_0, \bigcup_{i \geq 0} \text{urme}(P_i))$$

Pe viitor vom aplica operatorul de limită \bigsqcup numai proceselor care formează un lanț.

O ordine parțială se spune că este completă dacă are un cel mai mic element și toate lanțurile au o cea mai mică margine superioară. Mulțimea tuturor proceselor cu un alfabet A dat formează o relație de ordine parțială completă (o.p.c) dacă satisface legile

$$L4 \quad STOP_A \sqsubseteq P \quad \text{dacă } \alpha P = A$$

$$L5 \quad P \sqsubseteq \bigsqcup_{i \geq 0} P_i$$

$$L6 \quad (\forall i \geq 0. P_i \sqsubseteq Q) \Rightarrow (\bigsqcup_{i \geq 0} P_i) \sqsubseteq Q$$

Mai mult, definiția lui μ (2.8.1 D5) poate fi reformulată în termeni de limită

$$L7 \quad \mu X:A. F(X) = (\bigsqcup_{i \geq 0} F(STOP_A^i))$$

O funcție F de la o opc la alta (sau aceeași) se spune că este continuă dacă se distribuie limitelor tuturor șirurilor

$$F(\bigsqcup_{i \geq 0} P_i) = \bigsqcup_{i \geq 0} F(P_i) \quad \text{dacă } \{P_i \mid i \geq 0\} \text{ este lanț}$$

(Toate funcțiile continue sunt monotone în sensul că

$$P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q) \quad \text{pentru toți } P \text{ și } Q$$

astfel că membrul drept al ecuației anterioare este de asemenea limita unui șir crescător). O funcție G de mai multe argumente este definită drept continuă dacă este continuă în fiecare din argumente separat, de exemplu

$$G((\bigsqcup_{i \geq 0} P_i), Q) = \bigsqcup_{i \geq 0} G(P_i, Q) \quad \text{pentru toți } Q$$

$$\text{și } G(Q, (\bigsqcup_{i \geq 0} P_i)) = \bigsqcup_{i \geq 0} G(Q, P_i) \quad \text{pentru toți } Q$$

Compunerea funcțiilor continue ne dă o funcție continuă. Într-adevăr, orice expresie construită prin aplicarea unor funcții continue la orice număr sau combinație a variabilelor este continuă în fiecare din acele variabile. De exemplu, dacă G , F și H sunt continue

$$G(F(X), H(X, Y))$$

este continuă în X deoarece

$$G(F(\bigsqcup_{i \geq 0} P_i), H((\bigsqcup_{i \geq 0} P_i), Y)) = \bigsqcup_{i \geq 0} G(F(P_i), H(P_i, Y)) \quad \text{pentru toți } Y$$

Toți operatorii (cu excepția lui \wedge) definiți în D3 - D7 sunt continui în sensul definit mai sus

$$L8 \quad (x:B \rightarrow (\bigsqcup_{i \geq 0} P_i(x))) = \bigsqcup_{i \geq 0} (x:B \rightarrow P_i(x))$$

$$L9 \quad \mu X:A.F(X, (\bigsqcup_{i \geq 0} P_i)) = \bigsqcup_{i \geq 0} \mu X:A.F(X, P_i) \quad \text{dacă } F \text{ este continuă}$$

$$L10 \quad (\bigsqcup_{i \geq 0} P_i) \parallel Q = Q \parallel (\bigsqcup_{i \geq 0} P_i) = \bigsqcup_{i \geq 0} (Q \parallel P_i)$$

$$L11 \quad f(\bigsqcup_{i \geq 0} P_i) = \bigsqcup_{i \geq 0} f(P_i)$$

Prin urmare dacă $F(X)$ este orice expresie construită numai în baza acestor operatori va fi continuă în X . Acum este posibil să demonstrăm teorema de punct fix

$$F(\mu X:A.F(X)) = F(\bigsqcup_{i \geq 0} F^i(STOP_A)) \quad \text{definiția lui } \mu$$

$$= \bigsqcup_{i \geq 0} F(F^i(STOP_A)) \quad \text{continuitatea lui } F$$

$$= \bigsqcup_{i \geq 1} F^i(STOP_A) \quad \text{definiția lui } F^{i+1}$$

$$= \bigsqcup_{i \geq 0} F^i(STOP_A) \quad STOP_A \sqsubseteq F(STOP_A)$$

$$= \mu X:A.F(X) \quad \text{definiția lui } \mu$$

Această demonstrație s-a bazat numai pe faptul că F este continuă. Existența unei gărzi a lui F este necesară pentru a stabili unicitatea soluției.

2.8.3 Unicitatea soluțiilor

În acest paragraf vom trata mai formal raționamentul dat în paragraful 1.1.2 pentru a arăta că o ecuație definind un proces printr-o recursivitate cu gardă are o singură soluție. Astfel, vom face mai explicite condițiile generale pentru unicitatea unor astfel de soluții. Pentru ușurință ne ocupăm de ecuații simple. Metoda se poate ușor extinde la sisteme de ecuații.

Dacă P este un proces și n este un număr natural, definim $(P \upharpoonright n)$ ca un proces care se comportă ca P pentru primele n evenimente și apoi se oprește. Formal

$$(A, S) \upharpoonright n = (A, \{s \mid s \in S \wedge \#s \leq n\})$$

Urmează că

$$\text{L1 } P \upharpoonright 0 = \text{STOP}$$

$$\text{L2 } P \upharpoonright n \sqsubseteq P \upharpoonright (n+1) \sqsubseteq P$$

$$\text{L3 } P = \bigsqcup_{n \geq 0} (P \upharpoonright n)$$

$$\text{L4 } \bigsqcup_{n \geq 0} P_n = \bigsqcup_{n \geq 0} (P_n \upharpoonright n)$$

Fie F o funcție monotonă definită și cu valori în procese. F se spune că este constructivă dacă

$$F(X) \upharpoonright (n+1) = F(X \upharpoonright n) \upharpoonright (n+1) \quad \text{pentru toți } X$$

Aceasta înseamnă că comportarea lui $F(X)$ în primii $n+1$ pași este determinată numai de comportarea lui X în primii n pași. Astfel dacă $s \neq \diamond$

$$s \in \text{urme}(F(X)) \iff s \in \text{urme}(F(X \upharpoonright (\#s-1)))$$

Prefixul este un prim exemplu de funcție constructivă, deoarece

$$(c \rightarrow P) \upharpoonright (n+1) = (c \rightarrow (P \upharpoonright n)) \upharpoonright (n+1)$$

Alegerea generală este de asemenea constructivă

$$(x.B \rightarrow P(x)) \upharpoonright (n+1) = (x.B \rightarrow (P(x) \upharpoonright n)) \upharpoonright (n+1)$$

Funcția identitate I nu este constructivă deoarece

$$\begin{aligned} I \upharpoonright (c \rightarrow P) \upharpoonright 1 &= c \rightarrow \text{STOP} \\ &\neq \text{STOP} \\ &= I((c \rightarrow P) \upharpoonright 0) \upharpoonright 1 \end{aligned}$$

Putem acum formula teorema fundamentală

L5 Fie F o funcție constructivă. Ecuația

$$X = F(X)$$

are o singură soluție pentru X .

Demonstrație. Fie X o soluție arbitrară
Demonstrăm mai întâi prin inducție lema

$$X \upharpoonright n = F^n(STOP) \upharpoonright n$$

Avem $X \upharpoonright 0 = STOP = STOP \upharpoonright 0 = F^0(STOP) \upharpoonright 0$

Inducția

$$\begin{aligned} X \upharpoonright (n+1) &= F(X) \upharpoonright (n+1) && \text{deoarece } X = F(X) \\ &= F(X \upharpoonright n) \upharpoonright (n+1) && F \text{ este constructivă} \\ &= F(F^n(STOP) \upharpoonright n) \upharpoonright (n+1) && \text{ipoteză} \\ &= F(F^n(STOP)) \upharpoonright (n+1) && F \text{ este constructivă} \\ &= F^{n+1}(STOP) \upharpoonright (n+1) && \text{definiția } F^n \end{aligned}$$

Acum revenim la teorema dată

$$X = \bigsqcup_{n \geq 0} (X \upharpoonright n) \quad \text{L3}$$

$$= \bigsqcup_{n \geq 0} F^n(STOP) \upharpoonright n \quad \text{demonstrația de mai sus}$$

$$= \bigsqcup_{n \geq 0} F^n(STOP) \quad \text{L4}$$

$$= \mu X. (X) \quad \text{2.8.2 L7}$$

Deci toate soluțiile lui $X = F(X)$ sunt egale cu $\mu X. F(X)$; cu alte cuvinte $\mu X. F(X)$ este singura soluție a ecuației. \square

Utilitatea acestei teoreme este mult mai mare dacă putem recunoaște clar care funcții sunt constructive și care nu. Să definim o funcție nedestructivă G care satisface

$$G(P) \upharpoonright n = G(P \upharpoonright n) \upharpoonright n \quad \text{pentru toți } n \text{ și } P$$

Transformarea alfabetului este nedestructivă în acest sens, deoarece

$$f(P) \upharpoonright n = f(P \upharpoonright n) \upharpoonright n$$

La fel este și funcția identitate. Orice funcție monotonă care este constructivă este de asemenea nedestructivă. Dar operatorul *după* este destructiv, deoarece

$$\begin{aligned} ((c \rightarrow c \rightarrow STOP) \uparrow \langle c \rangle) \uparrow 1 &= c \rightarrow STOP \\ &\neq STOP \\ &= (c \rightarrow STOP) \uparrow \langle c \rangle \\ &= (((c \rightarrow c \rightarrow STOP) \uparrow 1) \uparrow \langle c \rangle) \uparrow 1 \end{aligned}$$

Orice compunere de funcții nedestructive este de asemenea nedestructivă (G și H), din cauză că

$$G(H(P)) \uparrow n = G(H(P) \uparrow n) \uparrow n = G(H(P \uparrow n) \uparrow n) \uparrow n = G(H(P \uparrow n)) \uparrow n$$

Mai important, orice compunere a unei funcții constructive cu una nedestructivă este constructivă. Dacă F , G , ..., H sunt nedestructive și una dintre ele este constructivă atunci

$$F(G(\dots(H(X))\dots))$$

este o funcție constructivă în X .

Raționamentele de mai sus se extind ușor la funcții de mai multe argumente. De exemplu compunerea concurentă este nedestructivă (în ambele argumente) deoarece

$$(P \parallel Q) \uparrow n = ((P \uparrow n) \parallel (Q \uparrow n)) \uparrow n$$

Fie E o expresie conținând variabila de proces X . Atunci E se spune că este cu gardă în X dacă orice apariție a lui X în E se face printr-o funcție constructivă aplicată variabilei și nu una destructivă. Astfel următoarea expresie este cu gardă în X

$$(c \rightarrow X) d \rightarrow f(X \parallel P) \mid c \rightarrow (f(x) \parallel Q) \parallel ((d \rightarrow X) \parallel R)$$

Cea mai importantă consecință este că proprietatea de constructivitate poate fi definită sintactic din următoarele condiții pentru gardă

D0 O expresie construită numai cu ajutorul operatorilor concurență, schimbare de simbol și alegere generală se spune că este păstrătoare de gardă.

D1 O expresie care nu conține pe X se spune că este cu gardă în X .

D2 O alegere generală

CONCURENȚĂ

171

$$(x:B \rightarrow P(X,x))$$

este cu gardă în X dacă $P(X, x)$ este păstrătoare de gardă pentru toți x .

D3 O schimbare de simbol $f(P(X))$ este cu gardă în X dacă $P(X)$ este cu gardă în X .

D4 Un sistem concurent $P(X) \parallel Q(X)$ este cu gardă în X dacă atât $P(X)$ cât și $Q(X)$ sunt cu gardă în X .

În final ajungem la concluzia

L6 Dacă E este cu gardă în X atunci ecuația

$$X=E$$

are soluție unică.

3 Nedeterminism

3.1 Introducere

Operatorul alegere ($x:B \rightarrow P(x)$) definește un proces care poate avea mai multe comportări posibile. Operatorul concurență \parallel permite altui proces să facă o selecție între alternativele oferite de mulțimea B . De exemplu, automatul de dat rest *SCH5C* (1.1.3 X2) oferă clientului alegerea restului cu trei monede mici și una mare sau două mari și una mică. Astfel de procese se numesc *deterministe*, din cauză că ori de câte ori există mai mult de un eveniment, alegerea între ele este controlată din exterior, din mediul în care se află procesul. Alegerea este determinată atât în sensul că mediul o poate face el însuși sau în sensul mai slab că mediul poate observa care alegere a fost făcută chiar în momentul când se face ea.

Uneori un proces are o gamă de comportări posibile dar mediul procesului nu are nici o posibilitate să influențeze sau chiar să observe selecția între alegerile făcute. De exemplu, un alt automat de dat rest poate da rest în orice situație de mai sus, dar alegerea între ele nu poate fi controlată sau chiar prezisă de client. Alegerea este făcută intern de mașina însăși într-un mod arbitrar sau nedeterminist. Mediul nu poate controla alegerea sau măcar să o observe, nici nu poate să depisteze când a fost făcută alegerea cu toate că poate să-și dea seama mai târziu care alegere a fost făcută din comportarea ulterioară a procesului.

Nu este nimic misterios cu acest fel de nedeterminism. El provine dintr-o decizie deliberată de a ignora factorii care influențează selecția. De exemplu, combinația de rest dată de mașină poate depinde într-un fel de încărcarea cu monezi mari și mici, însă aceste evenimente au fost excluse din alfabet. Astfel nedeterminismul este util pentru a menține un nivel înalt de abstractizare în descrierile comportării sistemelor.

3.2 SAU nedeterminist

Dacă P și Q sunt procese, atunci introducem notația

$$P \sqcap Q \quad (P \text{ sau } Q)$$

care înseamnă un proces ce se comportă atât ca P cât și ca Q , unde selecția între ele este făcută arbitrar fără cunoștința sau controlul mediului extern. Alfabetele operanzilor se presupun a fi aceleași.

$$\alpha(P \sqcap Q) = \alpha P = \alpha Q$$

Exemple

X1 Un automat de rest cu două combinații

$$SCH5D = (in5p \rightarrow (ex1p \rightarrow ex1p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5D) \sqcap (ex2p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5D))$$

□

X2 $SCH5D$ poate da o combinație diferită de rest la fiecare utilizare. Iată în continuare un exemplu de automat care totdeauna dă aceeași combinație, dar nu știm inițial care va fi aceea (vezi 1.1.2 X3, X4).

$$SCH5E = SCH5A \sqcap SCH5B$$

Desigur, după ce automatul dă prima monedă comportarea ulterioară este complet previzibilă. Din acest motiv,

$$SCH5D \neq SCH5E$$

□

Nedeterminismul a fost introdus aici în forma sa cea mai pură și simplă de operatorul binar \sqcap . Desigur însă, \sqcap nu este prea util în implementarea unui proces. Ar fi o prostie să construim atât P cât și Q , să-i punem într-o căciulă, să facem o alegere arbitrară între ei și să aflăm cealaltă alternativă. Principalul avantaj al nedeterminismului este în *specificarea* unui proces. Un proces specificat ca $(P \sqcap Q)$ poate fi implementat atât prin construirea lui P cât și a lui Q . Alegerea poate fi făcută în avans de cel care implementează pe motive nerelevante (și deliberat ignorate) în specificare, ca de exemplu cost minim, timpi rapizi de răspuns. De fapt operatorul \sqcap nu va fi folosit prea des nici în specificații. Nedeterminismul provine mai natural din definirea altor operatori prezentați mai departe în acest capitol.

3.2.1 Legi

Legile algebrice care guvernează alegerea nedeterministă sunt foarte simple și evidente. Alegerea între P și P dă tot P .

$$L1 \quad P \sqcap P = P \quad (\text{idempotența})$$

Nu contează în ce ordine se prezintă alegerea.

$$L2 \quad P \sqcap Q = Q \sqcap P \quad (\text{simetria})$$

O alegere între trei procese poate fi împărțită în două alegeri binare. Nu contează în ce fel se face aceasta.

$$L3 \quad P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \quad (\text{asociativitate})$$

Momentul în care se face alegerea nedeterministă nu este important. Un proces care întâi se angajează în x și apoi face o alegere nedeterministă este la fel cu unul care întâi face alegerea și apoi se angajează în x .

$$L4 \quad x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q) \quad (\text{distributivitate})$$

Legea L4 afirmă că operatorul prefix se distribuie în nedeterminism. Astfel de operatori se spune că sunt *distributivi*. Un operator diadic se spune că este distributiv dacă se distribuie cu \sqcap independent în ambele argumente. Cei mai mulți din operatorii definiți până acum pentru procese sunt distributivi în acest sens.

$$L5 \quad (x.B \rightarrow (P(x) \sqcap Q(x))) = (x.B \rightarrow P(x)) \sqcap (x.B \rightarrow Q(x))$$

$$L6 \quad P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$$

$$L7 \quad (P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$$

$$L8 \quad f(P \sqcap Q) = f(P) \sqcap f(Q)$$

Totuși operatorul de recursivitate *nu* este distributiv cu excepția cazului trivial când operandii lui \sqcap sunt identici. Aceasta se arată simplu prin diferența între cele două procese.

$$P = \mu X. ((a \rightarrow X) \sqcap (b \rightarrow X))$$

$$Q = (\mu X. (a \rightarrow X)) \sqcap (\mu X. (b \rightarrow X))$$

P poate face o alegere independentă între a și b la fiecare iterație astfel că urmele sale includ

$$\langle a, b, b, a, b \rangle$$

Q trebuie să facă o alegere între a și b tot timpul astfel că urmele sale nu includ pe cea scrisă mai sus. Totuși P poate alege să se angajeze în a sau b tot timpul astfel că

$$urme(Q) \leq urme(P)$$

În unele teorii, nedeterminismul este obligat să fie *echitabil* în sensul că un eveniment care *poate* apărea infinit de des *trebuie* să apară (cu toate că nu există nici o limită a cât de mult poate fi întârziat). În teoria noastră nu există un astfel de concept de echitate. Deoarece observăm numai secvențe finite ale comportării unui proces, dacă un eveniment poate fi amânat la infinit nu putem spune dacă va avea loc sau nu. Dacă vrem să insistăm ca evenimentul să apară trebuie să existe un număr n astfel că orice urmă mai mare ca n va conține acel eveniment. Procesul în care se întâmplă acest lucru trebuie proiectat explicit pentru a satisface această constrângere. De exemplu, în procesul P_0 definit mai jos impunem ca evenimentul a să apară în mai puțin de n pași de la apariția anterioară.

$$\begin{cases} P_i = (a \rightarrow P_0) \sqcap (b \rightarrow P_{i+1}) & \text{pentru } 0 \leq i < n \\ P_n = (a \rightarrow P_0) \end{cases}$$

Mai târziu vom vedea că atât Q cât și P_0 sunt implementări valide ale lui P .

Dacă se cere echitatea nedeterminismului aceasta poate fi specificată și implementată la un nivel separat, de exemplu, prin atribuirea probabilităților diferite de zero alternativelor unei alegeri nedeterminate. Pare rezonabil de a separa raționamentele probabilistice complexe de preocupările privind corectitudinea logică a comportării proceselor.

Datorită legilor L1–L3 este util de introdus un operator de alegere multiplă. Fie S o mulțime finită nevidă.

$$S = \{i, j, \dots, k\}$$

Atunci definim

$$\prod_{x \in S} P(x) = P(i) \sqcap P(j) \sqcap \dots \sqcap P(k)$$

$\prod_{x \in S}$ este fără sens când S este vidă sau infinită.

3.2.2 Implementări

Așa cum am arătat mai sus unul din principalele motive pentru care am introdus nedeterminismul este diminuarea detaliilor implementărilor. Aceasta înseamnă că pot fi mai multe implementări ale unui proces nedeterminist P fiecare cu un mod observabil și diferit de comportare. Diferențele provin din

diferitele decizii nedeterministe inerente lui \cdot . O anumită alegere poate fi făcută de cel care implementează înainte ca procesul să înceapă sau poate fi amânată până la derularea procesului.

De exemplu, o implementare a lui $(P \sqcap Q)$ ar fi selectarea primului operand

$$sau1(P, Q) = P$$

Altă implementare este obținută prin selectarea celui de-al doilea operand pe motive de eficiență mai mare pe o anumită mașină.

$$sau2(P, Q) = Q$$

În fine, o a treia implementare amână decizia până ce procesul începe să se execute. Apoi se permite mediului să facă alegerea, selectând un eveniment care este posibil pentru un proces dar nu și pentru celălalt. Dacă evenimentul este posibil pentru ambele procese decizia este iarăși amânată.

$$\begin{aligned} sau3(P, Q) = & \lambda x. \text{if } P(x) = \text{"BLIP"} \text{ then } Q(x) \\ & \text{else if } Q(x) = \text{"BLIP"} \text{ then } P(x) \\ & \text{else } sau3(P(x), Q(x)) \end{aligned}$$

Aici am dat trei posibile implementări ale aceluiași operator. De fapt sunt mult mai multe : de exemplu o implementare care se comportă ca *sau3* pentru primii 5 pași, iar dacă toți acești pași sunt posibili atât pentru *P* cât și pentru *Q* atunci se alege arbitrar *P*.

Deoarece proiectantul procesului $(P \sqcap Q)$ nu are controlul dacă *P* sau *Q* vor fi selectați, el trebuie să se asigure că acest sistem va funcționa corect în ambele cazuri. Dacă există vreun risc ca atât *P* cât și *Q* să se blocheze interacționând cu mediul atunci $(P \sqcap Q)$ are de asemenea acest risc. Implementarea *sau3* minimizează riscul de blocaj întârziind alegerea până ce o va face mediul și apoi selectând dintre *P* sau *Q* pe cel care nu se blochează. Din acest motiv definiția lui *sau3* este cunoscută ca *nedeterminism angelic*. Dar prețul plătit este mare în raport cu eficiența : dacă alegerea între *P* și *Q* nu este făcută în primul pas, atât *P* cât și *Q* trebuie să se execute concurrent până ce mediul alege un eveniment care este posibil pentru unul dar nu și pentru celălalt. În cazul simplu dar extrem *sau3(P, P)*, aceasta nu se va produce și ineficiența va fi evident maximă.

În contrast cu *sau3*, implementările *sau1* și *sau2* sunt asimetrice în sensul că

$$sau1(P, Q) \neq sau1(Q, P)$$

Aceasta pare să afecteze legea 3.2.1 L2, dar nu este așa. Legile se aplică proceselor, nu unei implementări particulare a lor. De fapt ele afirmă identitatea mulțimii tuturor implementărilor membrilor lor stângi și drepi. De exemplu, deoarece *sau3* este simetric

$$\begin{aligned}\{sau1(P,Q), sau2(P,Q), sau3(P,Q)\} &= \{P,Q, sau3(P,Q)\} \\ &= \{sau2(Q,P), sau1(Q,P), sau3(Q,P)\}\end{aligned}$$

Unul din avantajele introducerii nedeterminismului este evitarea pierderii de simetrie care ar rezulta din selectarea uneia din cele două implementări simple și totodată evitarea ineficienței unei implementări simetrice gen *sau3*.

3.2.3 Urme

Dacă *s* este o urmă a lui *P* atunci *s* este o urmă posibilă pentru $(P \sqcap Q)$ în cazul că este selectat *P*. Similar dacă *s* este o urmă a lui *Q*, este de asemenea o urmă a lui $(P \sqcap Q)$. Reciproc, fiecare urmă a lui $(P \sqcap Q)$ trebuie să fie o urmă a uneia din cele două alternative. Comportarea lui $(P \sqcap Q)$ după *s* este definită de oricare din *P* sau *Q* s-ar putea angaja în *s*. Dacă ambele ar putea, alegerea rămâne nedeterministă.

$$L1 \quad urme(P \sqcap Q) = urme(P) \cup urme(Q)$$

$$\begin{aligned}L2 \quad (P \sqcap Q) \backslash s &= Q \backslash s && \text{dacă } s \in (urme(Q) - urme(P)) \\ &= P \backslash s && \text{dacă } s \in (urme(P) - urme(Q)) \\ &= (P \backslash s) \sqcap (Q \backslash s) && \text{dacă } s \in (urme(P) \cap urme(Q))\end{aligned}$$

3.3 Alegere generală

Mediul în care evoluează $(P \sqcap Q)$ nu are controlul sau chiar cunoștința de alegerea care este făcută între *P* și *Q* sau a momentului în care se face alegerea. Astfel $(P \sqcap Q)$ nu este o metodă bună de a combina procese din cauză că mediul trebuie să fie pregătit să trateze atât cu *P* cât și cu *Q*, cu unul dintre ele separat fiind mult mai ușor de tratat. De aceea introducem o altă operație, $(P \sqcup Q)$, pentru ca mediul să poată controla care dintre *P* și *Q* va fi selectat, asigurându-se exercitarea controlului cu prima ocazie. Dacă evenimentul *nu* este un prim eveniment posibil pentru *P* atunci va fi selectat *Q*, iar dacă *Q* nu se poate angaja inițial în eveniment va fi selectat *P*. Dacă totuși primul eveniment este posibil atât pentru *P* cât și pentru *Q* atunci alegerea între cele două procese este nedeterministă. (Desigur, dacă evenimentul este imposibil atât pentru *P* cât și pentru *Q*, el nu poate să apară). Ca de obicei

$$\alpha(P \sqcup Q) = \alpha P = \alpha Q$$

În cazul când nici un eveniment inițial pentru P nu este posibil pentru Q , operatorul de alegere generală este același cu $|$ care a fost folosit pentru a reprezenta alegerea între două evenimente diferite

$$(c \rightarrow P \sqcup d \rightarrow Q) = (c \rightarrow P | d \rightarrow Q) \quad \text{dacă } c \neq d$$

Dacă totuși evenimentele inițiale sunt aceleași, $(P \sqcup Q)$ degenează în alegere nedeterministă

$$(c \rightarrow P \sqcup c \rightarrow Q) = (c \rightarrow P) \sqcap (c \rightarrow Q)$$

Aici am adoptat prioritatea lui \rightarrow față de \sqcup .

3.3.1 Legi

Legile algebrice pentru \sqcup sunt similare cu acelea pentru \sqcap iar motivele sunt aceleași

L1 - L3 \sqcup este idempotent, simetric și asociativ

L4 $P \sqcup STOP = P$

Următoarea lege formalizează definiția explicativă a operatorului

L5 $(x:A \rightarrow P(x)) \sqcup (y:B \rightarrow Q(y)) = (z:(A \cup B) \rightarrow (\text{if } z \in (A-B) \text{ then } P(z) \\ \text{else if } z \in (B-A) \text{ then } Q(z) \\ \text{else if } z \in (A \cap B) \text{ then } (P(z) \sqcap Q(z))))$

Ca toți ceilalți operatori introduși până acum (în afară de recursivitate), \sqcup distribuie față de \sqcap

L6 $P \sqcup (Q \sqcap R) = (P \sqcup Q) \sqcap (P \sqcup R)$

Mai surprinzător este că \sqcap distribuie față de \sqcup

L7 $P \sqcap (Q \sqcup R) = (P \sqcap Q) \sqcup (P \sqcap R)$

Această lege afirmă că alegerile nedeterminate și cele făcute de mediu sunt independente în sensul că selecția făcută într-un fel nu este influențată de cealaltă parte. Fie John agentul care face alegerile nedeterminate și fie Mary me-

diul. În partea stângă a legii, John alege (\sqcap) între P și a o lăsa pe Mary să aleagă între Q și R . În partea dreaptă Mary are posibilitatea

atât (1) să-i ofere lui John alegerea între P și Q
cât și (2) să-i ofere lui John alegerea între P și R .

În ambele părți ale ecuației, dacă John alege P , atunci P va fi alegerea generală. Dacă John nu-l selectează pe P alegerea între Q și R este făcută de Mary. Astfel rezultatele alegerilor descrise în partea stângă și dreaptă a legii sunt totdeauna egale. Desigur, același raționament se aplică la L6.

Explicația dată mai sus este într-un fel subtilă. Desigur se va înțelege mai bine legea ca o consecință inevitabilă a altor legi și definiții mai evidente pe care le vom da în continuarea acestui capitol.

3.3.2 Implementare

Implementarea operatorului alegere generală rezultă imediat din legea L5. Presupunând simetria lui *sau*, este de asemenea simetrică funcția

$$\begin{aligned} \text{alternat}(P, Q) = \lambda x. & \text{ if } P(x) = \text{"BLIP"} \text{ then } Q(x) \\ & \text{ else if } Q(x) = \text{"BLIP"} \text{ then } P(x) \\ & \text{ else } \text{sau}(P(x), Q(x)) \end{aligned}$$

3.3.3 Urme

Fiecare urmă a lui $(P \sqcap Q)$ trebuie să fie o urmă a lui P sau a lui Q și reciproc

$$\text{L1 } \text{urme}(P \sqcap Q) = \text{urme}(P) \cup \text{urme}(Q)$$

Următoarea lege este puțin diferită față de legea corespunzătoare pentru \sqcap

$$\begin{aligned} \text{L2 } (P \sqcap Q)/s &= P/s & \text{dacă } s \in \text{urme}(P) - \text{urme}(Q) \\ &= Q/s & \text{dacă } s \in \text{urme}(Q) - \text{urme}(P) \\ &= (P/s) \sqcap (Q/s) & \text{dacă } s \neq \diamond \text{ și } s \in \text{urme}(P) \cap \text{urme}(Q) \end{aligned}$$

3.4 Refuzuri

Distincția între $(P \sqcap Q)$ și $(P \sqcup Q)$ este subtilă. Ele nu se pot distinge prin urmele lor din cauză că fiecare urmă a unuia este o posibilă urmă a celuilalt. Totuși este posibil să le punem într-un mediu în care $(P \sqcap Q)$ se poate bloca la primul pas, în timp ce $(P \sqcup Q)$ nu. În următorul exemplu fie $x \neq y$ și

$$P=(x \rightarrow P), Q=(y \rightarrow Q), \alpha P = \alpha Q = \{x, y\}$$

$$\text{Apoi } (P \sqcap Q) \parallel P = (x \rightarrow P) \\ = P$$

$$\text{dar } (P \sqcap Q) \parallel P = (P \parallel P) \sqcap (Q \parallel P) \\ = P \sqcap \text{STOP}$$

Aceasta ne arată că în mediul P , $(P \sqcap Q)$ poate ajunge la blocaj dar $(P \sqcap Q)$ nu. Desigur, chiar cu $(P \sqcap Q)$ nu putem fi siguri că nu va apare blocaj. Dacă nu apare, nu vom ști niciodată dacă ar fi putut apărea. Numai simpla posibilitate a apariției blocajului și este suficientă pentru a distinge $(P \sqcap Q)$ de $(P \sqcap Q)$.

În general, fie X o mulțime de evenimente oferite inițial de mediu unui proces P , care în acest context are același alfabet cu alfabetul lui P . Dacă este posibil pentru P să se blocheze în primul pas când este plasat în acest mediu, spunem că X este un *refuz* al lui P . Mulțimea tuturor acestor refuzuri ale lui P este notată

$$\text{refuzuri}(P)$$

De remarcat că refuzurile unui proces constituie o familie de mulțimi de simboluri. Aceasta este o complexitate nefericită, dar pare să fie de neînlăturat într-o manieră adecvată de tratare a nedeterminismului. În loc de refuzuri ar fi mai natural să folosim mulțimea de simboluri pe care un proces este *gata* să le accepte, totuși refuzurile sunt mai simple din cauză că se supun legilor L9 și L10 din paragraful 3.4.1 (mai jos) în timp ce legile corespunzătoare pentru mulțimile *gata* ar fi mai complicate.

Introducerea conceptului de refuz permite să fie făcută o distincție clară și formală între procesele deterministe și nedeterministe. Un proces se spune că este *determinist* dacă nu refuză nici un eveniment în care se poate angaja. Cu alte cuvinte, o mulțime este un refuz pentru un proces determinist numai dacă mulțimea nu conține nici un eveniment în care procesul să se poată angaja inițial, sau mai formal

$$P \text{ este determinist} \Rightarrow (X \in \text{refuzuri}(P) \Rightarrow (X \cap P^0 = \{\})) \\ \text{unde } P^0 = \{x \mid \langle x \rangle \in \text{urme}(P)\}$$

Această condiție se aplică nu numai la pasul inițial al lui P dar de asemenea după orice secvență posibilă de evenimente pentru P . Astfel putem defini

$$P \text{ este determinist} \equiv \forall s: \text{urme}(P). (X \in \text{refuzuri}(P/s) \Rightarrow (X \cap (P/s)^0 = \{\}))$$

Un proces nedeterminist nu suferă de această proprietate, astfel că dacă există în orice moment cel puțin un eveniment în care să se poată angaja, de asemenea (ca rezultat al unei alegeri interne nedeterminate) poate refuza angajarea în acest eveniment chiar dacă mediul său cu care interacționează este pregătit să se angajeze.

3.4.1 Legi

Următoarele legi definesc refuzurile pentru diverse procese simple. Procesul *STOP* nu face nimic și refuză totul.

L1 $\text{refuzuri}(\text{STOP}_A) = \text{toate submulțimile lui } A \text{ (inclusiv } A)$

Un proces $(c \rightarrow P)$ refuză orice mulțime care nu conține evenimentul c

L2 $\text{refuzuri}(c \rightarrow P) = \{X \mid X \subseteq (\alpha P - \{c\})\}$

Aceste două legi se generalizează în

L3 $\text{refuzuri}(x:B \rightarrow P(x)) = \{X \mid X \subseteq (\alpha P - B)\}$

Dacă P poate refuza X la fel va face și $(P \sqcap Q)$ dacă este selectat P . Similar orice refuz al lui Q este de asemenea un posibil refuz al lui $(P \sqcap Q)$. Acestea sunt singurele refuzuri astfel că

L4 $\text{refuzuri}(P \sqcap Q) = \text{refuzuri}(P) \cup \text{refuzuri}(Q)$

O argumentație inversă se aplică la $(P \sqcup Q)$. Dacă X nu este un refuz al lui P , atunci P nu poate refuza pe X și deci nici $(P \sqcup Q)$. Similar dacă X nu este un refuz al lui Q , nu este un refuz nici al lui $(P \sqcup Q)$. Totuși, dacă atât P cât și Q pot refuza pe X , atunci și $(P \sqcup Q)$ poate.

L5 $\text{refuzuri}(P \sqcup Q) = \text{refuzuri}(P) \cap \text{refuzuri}(Q)$

Compararea lui L5 cu L4 ne dă diferența între \sqcap și \sqcup .

Dacă P poate refuza X și Q poate refuza Y atunci combinația lor $(P \parallel Q)$ poate refuza toate evenimentele refuzate de P precum și toate evenimentele refuzate de Q , cu alte cuvinte poate refuza reuniunea mulțimilor X și Y .

L6 $\text{refuzuri}(P \parallel Q) = \{X \cup Y \mid X \in \text{refuzuri}(P) \wedge Y \in \text{refuzuri}(Q)\}$

Pentru schimbarea de simbol legea este clară

$$L7 \quad \text{refuzuri}(f(P)) = \{f(X) \mid X \in \text{refuzuri}(P)\}$$

Există un număr de legi generale privind refuzurile. Un proces poate refuza numai evenimentele din propriul său alfabet. Un proces se blochează când mediul nu-i oferă evenimente. Dacă un proces refuză o mulțime nevidă el poate refuza orice submulțime a acelei mulțimi. În fine, orice eveniment x care nu apare ca refuz inițial poate fi adăugat la orice mulțime X deja refuzată.

$$L8 \quad X \in \text{refuzuri}(P) \Rightarrow X \subseteq \alpha P$$

$$L9 \quad \{\} \in \text{refuzuri}(P)$$

$$L10 \quad (X \cup Y) \in \text{refuzuri}(P) \Rightarrow X \in \text{refuzuri}(P)$$

$$L11 \quad X \in \text{refuzuri}(P) \Rightarrow (X \cup \{x\}) \in \text{refuzuri}(P) \vee \langle x \rangle \in \text{urme}(P) \text{ dacă } x \in \alpha P$$

3.5 Mascare (ascundere)

În general, alfabetul unui proces conține numai acele evenimente care se consideră a fi relevante iar aparițiile acestor evenimente necesită participarea simultană a mediului. În descrierea comportării interne a unui sistem adeseori avem nevoie să considerăm evenimente reprezentând tranziții interne aceluși sistem. Astfel de evenimente pot exprima interacțiunile și comunicațiile între componentele active concurente din care este construit sistemul, de exemplu LAN72 (2.6.X4) și 2.6.2.X3. După construirea sistemului, mascăm structura componentelor. De asemenea dorim să mascăm toate aparițiile acțiunilor interne sistemului. De fapt vrem ca aceste acțiuni să apară automat și instantaneu cât mai curând posibil fără a fi observate sau controlate de mediul procesului. Dacă C este o mulțime finită de evenimente gata de a fi mascate în acest mod, atunci

$$P \setminus C$$

este un proces care se comportă ca P cu excepția faptului că orice apariție a unui eveniment din C este mascată. De fapt intenția este ca

$$\alpha(P \setminus C) = (\alpha P) - C$$

Exemple

X1 Un automat de vândut zgomotos (2.3.X1) poate fi plasat într-o carcasă antifonică

$$AVSUNET \setminus \{\text{bing}, \text{bang}\}$$

Poate fi înlăturată din alfabet și posibilitatea de a livra bomboane fără a afecta comportarea actuală. Procesul care rezultă este automatul simplu

$$A1'S=AVSUNET\{bing,bang,bonbon\}$$

□

Când două procese au fost combinate pentru a funcționa concurrent, interacțiunile lor comune sunt de obicei privite ca activități interne sistemului rezultat. Ele se preconizează să apară autonom și cât mai repede posibil fără cunoștința sau intervenția mediului exterior sistemului. Astfel, simbolurile din intersecția celor două alfabete ale componentelor trebuie să fie mascate.

$$X2 \quad \text{Fie } \alpha P=\{a,c\}, \alpha Q=\{b,c\}, P=(a \rightarrow c \rightarrow P), Q=(c \rightarrow b \rightarrow Q) \quad (2.3.1.X1)$$

Acțiunea c în alfabetele lui P și Q este acum privită ca o acțiune internă ce trebuie mascată

$$\begin{aligned} (P\|Q)\setminus\{c\} &= (a \rightarrow c \rightarrow \mu X.(a \rightarrow b \rightarrow c \rightarrow \lambda \setminus b \rightarrow a \rightarrow c \rightarrow \lambda))\setminus\{c\} \\ &= a \rightarrow \mu X.(a \rightarrow b \rightarrow \lambda \setminus b \rightarrow a \rightarrow \lambda) \end{aligned}$$

□

3.5.1 Legi

Primele legi stabilesc că mascarea nici unui simbol nu are efect și că nu este nici o diferență în ce ordine sunt mascate simbolurile unei mulțimi. Celelalte legi din acest grup ne arată cum mascarea se distribuie cu ceilalți operatori. Mascarea a nimic lasă procesul neschimbat.

$$L1 \quad P \setminus \{\} = P$$

Pentru a masca o mulțime de simboluri și ulterior altele este totuna cu a le masca simultan

$$L2 \quad (P \setminus B) \setminus C = P \setminus (B \cup C)$$

Mascarea se distribuie obișnuit cu alegerea nedeterministă

$$L3 \quad (P \sqcap Q) \setminus C = (P \setminus C) \sqcap (Q \setminus C)$$

Mascarea nu afectează un proces oprit ci numai alfabetul său

$$L4 \quad STOP_A \setminus C = STOP_{A-C}$$

Scopul mascării este de a permite oricărui eveniment mascat să apară imediat și independent, în condițiile invizibilității acestor apariții. Evenimentele nemascate rămân neschimbate.

$$\begin{aligned} \text{L5} \quad (x \rightarrow P) \setminus C &= x \rightarrow (P \setminus C) && \text{dacă } x \notin C \\ &= P \setminus C && \text{dacă } x \in C \end{aligned}$$

Dacă C conține numai evenimente în care P și Q participă independent mascarea lui C se distribuie în compunerea lor concurentă

$$\begin{aligned} \text{L6} \quad \text{Dacă } \alpha P \cap \alpha Q \cap C &= \{ \} \\ \text{atunci } (P \parallel Q) \setminus C &= (P \setminus C) \parallel (Q \setminus C) \end{aligned}$$

Legea de mai sus nu este prea utilă în mod normal din cauză că ceea ce dorim să mascăm sunt interacțiunile dintre procesele concurente, cu alte cuvinte mulțimea $\alpha P \cap \alpha Q$ a evenimentelor comune în care participă.

Mascarea se distribuie evident cu schimbarea de simbol dată de o funcție injectivă

$$\text{L7} \quad f(P \setminus C) = f(P) \setminus f(C)$$

Dacă nici unul din evenimentele posibile inițial ale alegerii nu este mascat atunci alegerea inițială rămâne aceeași ca și înainte de mascare.

$$\begin{aligned} \text{L8} \quad \text{Dacă } B \cap C &= \{ \} \\ \text{atunci } (x: B \rightarrow P(x)) \setminus C &= (x: B \rightarrow (P(x) \setminus C)) \end{aligned}$$

Ca și operatorul alegere generală \square , mascarea evenimentelor poate introduce nedeterminism. Când pot să apară mai multe evenimente mascate diferite nu este precizat care va apare, dar oricare va apare este mascat.

$$\text{L9} \quad \text{Dacă } B \subseteq C \text{ și } B \text{ este finit și nevid}$$

$$\text{atunci } (x: B \rightarrow P(x)) \setminus C = \prod_{x: B} (P(x) \setminus C)$$

Într-un caz intermediar când o parte din evenimentele inițiale sunt mascate și altele nu, situația este ceva mai complicată. Considerăm procesul

$$(c \rightarrow P \mid d \Rightarrow Q) \setminus C \quad \text{unde } c \in C, d \notin C$$

Evenimentul mascat c poate apărea imediat. În acest caz întreaga comportarea va fi definită de $(P \setminus C)$ și posibilitatea apariției evenimentului d va fi eliminată.

Dar nu putem presupune sigur că d nu va apărea. Dacă mediul este pregătit să o facă, d poate foarte bine apărea înaintea evenimentului mascat, după care evenimentul mascat c poate să nu mai apară. Dar chiar dacă d apare, ar putea fi angajat de procesul $(P \setminus C)$ după apariția mascată a lui c . În acest caz, comportarea ce rezultă este definită de

$$(P \setminus C) \sqcap (d \rightarrow (Q \setminus C))$$

Alegerea între membrul drept și $(P \setminus C)$ este nedeterministă. Desigur aceasta este o justificare cam sinuoasă pentru destul de complexa lege

$$(c \rightarrow P \mid d \rightarrow Q) \setminus c = (P \setminus C) \sqcap ((P \setminus C) \sqcap (d \rightarrow (Q \setminus c)))$$

Raționamente similare justifică legea mai generală

L10 Dacă $C \cap B \neq \{\}$ și este finită

atunci $(x: B \rightarrow P(x)) \setminus C = Q \sqcap (Q \sqcap (x: (B - C) \rightarrow (P(x) \setminus C)))$

unde $Q = \prod_{x: B \cap C} (P(x) \setminus C)$

O reprezentare a acestor legi este dată în paragraful 3.5.4.

De notat că $\setminus C$ nu se distribuie retroactiv cu \sqcap . Un contraexemplu este

$$\begin{aligned} (c \rightarrow STOP \sqcap d \rightarrow STOP) \setminus \{c\} &= STOP \sqcap STOP \sqcap (d \rightarrow STOP) && \text{L10} \\ &= STOP \sqcap (d \rightarrow STOP) && 3.3.1 \text{ L4} \\ &\neq d \rightarrow STOP \\ &= STOP \sqcap (d \rightarrow STOP) \\ &= ((c \rightarrow STOP) \setminus \{c\}) \sqcap ((d \rightarrow STOP) \setminus \{c\}) \end{aligned}$$

Mascarea reduce alfabetul unui proces. De asemenea putem defini o operație care extinde alfabetul unui proces prin includerea simbolurilor unei mulțimi B .

$$\begin{aligned} \alpha(P_{+B}) &= \alpha P \cup B \\ P_{+B} &= (P \parallel STOP_B) && \text{dacă } B \cap \alpha P = \{\} \end{aligned}$$

Nici unul din noile evenimente din B nu va apare de fapt, astfel că comportarea lui P_{+B} este efectiv aceeași cu aceea a lui P .

$$\text{L11 } \text{urme}(P_{+B}) = \text{urme}(P)$$

Prin urmare mascarea lui B este inversul extinderii alfabetului prin B .

$$L12 \quad (P_{+B}) \setminus B = P$$

Este cazul să ne punem aici o problemă ce va fi rezolvată mai târziu în paragraful 3.8. În cazuri simple, mascarea se distribuie cu recursivitatea

$$\begin{aligned} \mu X. A.(c \rightarrow X) \setminus \{c\} &= \mu X. (A - \{c\}).((c \rightarrow X_{+\{c\}}) \setminus \{c\}) \\ &= \mu X. (A - \{c\}).X \end{aligned} \quad \text{din L12, L5}$$

Astfel încercarea de a masca o secvență *infinită* de evenimente consecutive ne conduce la același rezultat nefavorabil ca și o buclă infinită sau o recursivitate fără gardă. Termenul general pentru acest lucru este *divergență*.

Aceeași problemă apare chiar dacă procesul divergent este capabil de a se angaja la infinit într-un eveniment nemascat, de exemplu

$$\begin{aligned} \mu X. (c \rightarrow X \sqcap d \rightarrow P) \setminus \{c\} &= \mu X. ((c \rightarrow X \sqcap d \rightarrow P) \setminus \{c\}) \\ &= \mu X. (X \setminus \{c\}) \sqcap ((X \setminus \{c\}) \sqcap d \rightarrow (P \setminus \{c\})) \end{aligned} \quad \text{din L10}$$

Din nou, recursivitatea este fără gardă și ne conduce la divergență. Chiar și așa, apare că mediul oferă infinit posibilitatea selectării lui d și nu există nici o metodă de a preveni procesul în a alege la infinit angajarea evenimentului mascat. Această posibilitate pare să conducă la realizarea celei mai mari eficiențe în implementare. De asemenea chestiunea este într-un fel legată de decizia noastră de a nu insista pe echitatea nedeterminismului, așa cum s-a prezentat în paragraful 3.2.1. O discuție mai riguroasă despre divergență se va face în paragraful 3.8.

Există un sens, unul important, în care mascarea este de fapt echitabilă. Fie $d \in \alpha R$ și considerăm procesul

$$\begin{aligned} &((c \rightarrow a \rightarrow P \mid d \rightarrow STOP) \setminus \{c\}) \parallel (a \rightarrow R) \\ &= ((a \rightarrow P \setminus \{c\}) \sqcap (a \rightarrow P \setminus \{c\} \sqcap d \rightarrow STOP)) \parallel (a \rightarrow R) \quad \text{din L10} \\ &= (a \rightarrow P \setminus \{c\}) \parallel (a \rightarrow R) \sqcap (a \rightarrow P \setminus \{c\} \sqcap d \rightarrow STOP) \parallel (a \rightarrow R) \\ &= a \rightarrow ((P \setminus \{c\}) \parallel R) \end{aligned}$$

Acest exemplu ne arată că un proces care oferă alegerea între o acțiune mascată c și una nemascată d , nu poate insista să apară acțiunea nemascată. Dacă mediul (în acest exemplu, $a \rightarrow R$) nu este pregătit pentru d , atunci evenimentul mascat trebuie să apară, astfel că mediul are șansa să interacționeze cu procesul care rezultă (în exemplu $(a \rightarrow P \setminus \{c\})$).

3.5.2 Implementare

Pentru simplitate vom implementa o operație care maschează un singur simbol odată

$$masc(P, c) = P \setminus \{c\}$$

O mulțime de simboluri pot fi mascate unul după altul

$$P \setminus \{c_1, c_2, \dots, c_n\} = (\dots ((P \setminus \{c_1\}) \setminus \{c_2\}) \dots) \setminus \{c_n\}$$

Cea mai simplă implementare este aceea care face totdeauna ca evenimentul mascat să apară invizibil ori de câte ori poate și cât mai curând posibil.

$$\begin{aligned} masc(P, c) = & \text{if } P(c) = \text{"BLIP"} \text{ then} \\ & (\lambda x. \text{if } P(x) = \text{"BLIP"} \text{ then "BLIP"} \\ & \quad \text{else } masc(P(x), c)) \\ & \text{else } masc(P(c), c) \end{aligned}$$

Să vedem ce se întâmplă când funcția de mascare este aplicată unui proces capabil de a se angaja într-o secvență infinită de evenimente mascate, de exemplu

$$masc(\mu X. (c \rightarrow X \square d \rightarrow P), c)$$

În acest caz, testul $(P(c) = \text{"BLIP"})$ va genera totdeauna *FALSE*, astfel că funcția *masc* va selecta mereu ramura *else*, apelându-se de aceea imediat recursiv. Nu există ieșire din această recursivitate, astfel că nu va apare comunicare cu lumea exterioară. Acesta este prețul pentru încercarea de a implementa un proces divergent.

Prezenta implementare a mascării nu se supune la L2. Într-adevăr, ordinea în care se maschează simbolurile este importantă, așa cum se arată în exemplul

$$P = (c \rightarrow STOP \mid d \rightarrow a \rightarrow STOP)$$

$$\begin{aligned} \text{Atunci } masc(masc(P, c), d) &= masc(masc(STOP, c), d) \\ &= STOP \end{aligned}$$

$$\begin{aligned} \text{și } masc(masc(P, d), c) &= masc(masc((a \rightarrow STOP), d), c) \\ &= (a \rightarrow STOP) \end{aligned}$$

Dar așa cum s-a explicat în paragraful 3.2.2, o implementare particulară a unui operator nedeterminist nu trebuie să se supună legilor. Este suficient ca ambele rezultate arătate mai sus să permită implementări ale aceluiași proces

$$P \setminus \{c, d\} = (STOP \sqcap (a \rightarrow STOP))$$

3.5.3 Urme

Dacă t este o urmă a lui P , urma corespunzătoare lui $P \setminus C$ este obținută simplu din t prin eliminarea tuturor aparițiilor simbolurilor din C . Reciproc, fiecare urmă a lui $P \setminus C$ trebuie să se obțină dintr-o astfel de urmă a lui P . De aceea formulăm

$$L1 \quad \text{urme}(P \setminus C) = \{t \upharpoonright (\alpha P - C) \mid t \in \text{urme}(P)\} \quad \text{când } \forall s: \text{urme}(P). \neg \text{diverge}(P/s, C)$$

Condiția $\text{diverge}(P, C)$ semnifică divergența imediată a lui P odată cu mascarea lui C , cu alte cuvinte P se poate angaja într-o secvență nemărginită de evenimente mascate. De aceea definim

$$\text{diverge}(P, C) = \forall n. \exists s: \text{urme}(P) \cap C^* . \#s > n$$

Corespunzător unei singure urme s a lui $P \setminus C$ pot fi mai multe urme t ale comportării posibile în care se angajează P și care nu pot fi diferențiate după mascare, adică $t \upharpoonright (\alpha P - C) = s$. Următoarea lege formulează că după parcurgerea urmei s nu se poate ști care din comportările ulterioare ale lui P vor defini comportarea ulterioară a lui $(P \setminus C)$.

$$L2 \quad (P - C)/s = \left(\prod_{t: T} P/t \right) \setminus C$$

unde $T = \text{urme}(P) \cap \{t \mid t \upharpoonright (\alpha P - C) = s\}$

în condițiile când T este finit și $s \in \text{urme}(P \setminus C)$

Aceste legi sunt restricționate la cazul când procesul nu diverge. Restricțiile nu sunt importante din cauză că divergența nu se intenționează a fi niciodată rezultatul definirii unui proces. Pentru o completă tratare a divergenței a se vedea paragraful 3.8.

3.5.4. Reprezentări

Alegerea nedeterministă poate fi reprezentată printr-un nod din care ies două sau mai multe arce neetichetate. Ajungând în acel nod un proces trece imper-

ceptibil de-a lungul unuia din arce, alegerea fiind nedeterministă. Astfel $P \sqcap Q$ este reprezentat în fig. 3.1.

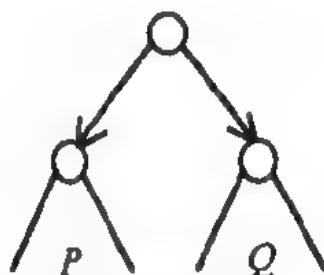


Figura 3.1

Legile algebrice ce guvernează nedeterminismul presupun identități între aceste reprezentări, de exemplu asociativitatea lui \sqcap este ilustrată în fig. 3.2.

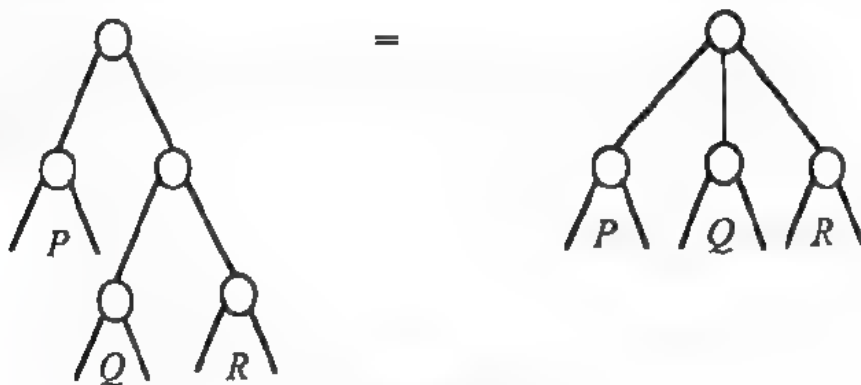


Figura 3.2

Mascarea simbolurilor poate fi privită ca o operație care pur și simplu elimină simbolurile respective din toate arcele pe care le etichetează astfel că aceste arce se transformă în arce neetichetate. Nedeterminismul rezultat decurge natural ca în fig. 3.3.

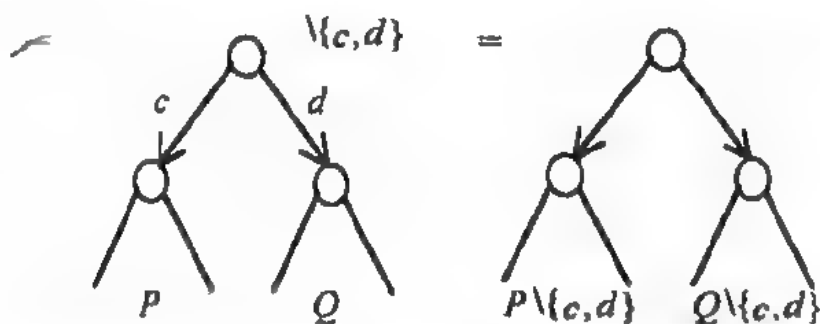


Figura 3.3

Dar care este semnificația unui nod când o parte din arcele sale sunt etichetate și altele nu? Răspunsul este dat de legea 3.5.1 L10. Un astfel de nod poate fi eliminat prin redesenare așa cum se arată în fig. 3.4.

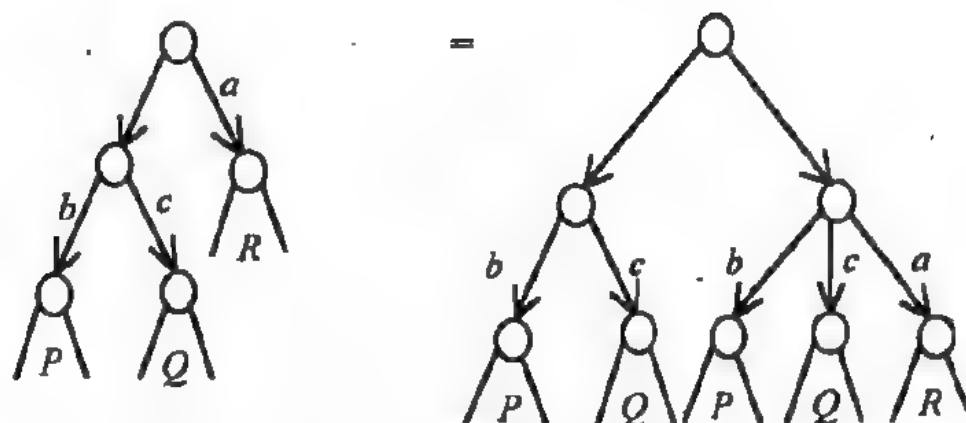


Figura 3.4

Este în general evident că astfel de eliminări sunt totdeauna posibile pentru arbori finiți. Ele sunt posibile și pentru grafuri infinite în condițiile când graful nu conține drumuri infinite din arce consecutive neetichetate ca de exemplu în fig. 3.5. O astfel de reprezentare apare numai în cazul divergenței pe care am decis deja să o privim ca o eroare.

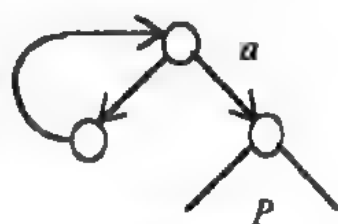


Figura 3.5

Ca un rezultat al aplicării transformării L10 este posibil ca un nod să poată dobândi două arce cu aceeași etichetă. Astfel de noduri pot fi eliminate cu legea dată la sfârșitul paragrafului 3.3 (fig. 3.6)

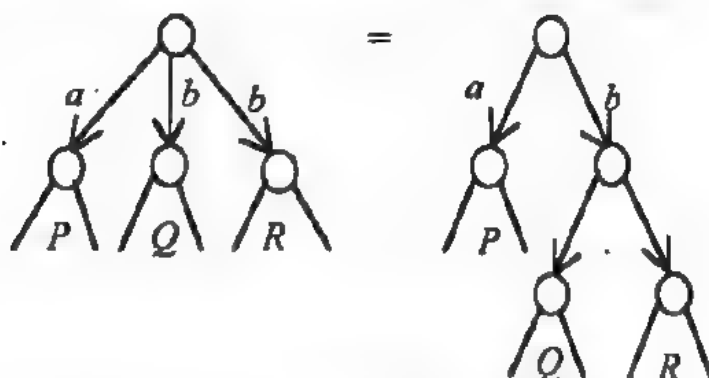


Figura 3.6

Reprezentările proceselor și legile care le guvernează sunt incluse aici ca un ajutor în memorizarea și înțelegerea lor. Aceste tehnici grafice nu se intenționează a fi utilizate în transformările sau manipulările cu procese extinse.

3.6 Întreșeserea

Operatorul \parallel definit în capitolul 2 necesită participarea simultană în acțiunile din alfabetul ambilor operanzi în timp ce celelalte acțiuni ale sistemului apar într-o întreșesere arbitrară. Utilizând acest operator este posibilă combinarea proceselor ce interacționează, având alfabete diferite, în sisteme ce prezintă activitate concurentă dar fără introducerea nedeterminismului.

Totuși este uneori util de a reuni procese cu același alfabet pentru a opera concurent fără interacțiunea sau sincronizarea lor directă. În acest caz, fiecare acțiune a sistemului este o acțiune exact a unui proces. Dacă unul dintre procese nu se poate angaja în acțiune atunci trebuie să o facă celălalt, dar dacă ambele procese s-ar putea angaja în acțiune, alegerea este nedeterministă. Această formă de compunere a proceselor este notată

$$P \parallel Q \quad (P \text{ întreșesut cu } Q)$$

și alfabetul său este definit normal

$$\alpha(P \parallel Q) = \alpha P = \alpha Q$$

Exemple

X1 Un AVS va accepta până la două monede înainte de a da până la două ciocolate (1.1.3 X6)

$$(AVS \parallel AVS) = AVS2$$

□

X2 Un valet format din patru valeți tineri subalterni, fiecare servind numai un singur filozof la un moment dat (a se vedea paragraful 2.5.3 și 2.6.4 X1)

$$E \parallel E \parallel E \parallel E$$

unde $E = VAL_T_M$

□

3.6.1 Legi

L1 - 3 \parallel este asociativă, simetrică și distributivă față de \sqcap

L4 $P \parallel STOP = P$

L5 $P \parallel RUN = RUN$

în condițiile în care P nu diverge

L6 $(x \rightarrow P) \parallel (y \rightarrow Q) = (x \rightarrow (P \parallel (y \rightarrow Q))) \sqcap y \rightarrow ((x \rightarrow P) \parallel Q)$

L7 Dacă $P = (x.A \rightarrow P(x))$

și $Q = (y.B \rightarrow Q(y))$

atunci $P|||Q = (x:A \rightarrow (P(x)|||Q)) \square (y:B \rightarrow (P|||Q(y)))$

De remarcat că $|||$ nu se distribuie cu \square . Aceasta este arătat prin următorul contraexemplu (unde $b \neq c$)

$$\begin{aligned} ((a \rightarrow STOP) ||| (b \rightarrow Q \square c \rightarrow R)) \gamma \langle a \rangle &= \\ &= (b \rightarrow Q \square c \rightarrow R) \\ &\neq ((b \rightarrow Q) \sqcap (c \rightarrow R)) \\ &= ((a \rightarrow STOP ||| b \rightarrow Q) \square (a \rightarrow STOP ||| c \rightarrow R)) \gamma \langle a \rangle \end{aligned}$$

În prima expresie din lanțul de egalități, apariția lui a poate să implice progresul numai operandului din stânga lui $|||$, astfel că nu există nedeterminism. Operandul din stânga se oprește și alegerea între b și c este lăsată la latitudinea mediului. În ultima expresie a lanțului de egalități, evenimentul a poate fi al oricărui operand al lui $|||$, de aceea alegerea este nedeterministă. Astfel mediul nu poate alege dacă următorul eveniment este b sau c .

L6 și L7 afirmă că mediul alege între evenimentele inițiale oferite de operandii lui $|||$. Nedeterminismul apare numai când evenimentul ales este posibil pentru ambii operanzi.

Exemplu

X1 Fie $R = (a \rightarrow b \rightarrow R)$

$$\begin{aligned} \text{Atunci } (R ||| R) &= (a \rightarrow ((b \rightarrow R) ||| R) \square a \rightarrow (R ||| (b \rightarrow R))) && \text{L6} \\ &= a \rightarrow ((b \rightarrow R) ||| R) \sqcap (R ||| (b \rightarrow R)) \\ &= a \rightarrow ((b \rightarrow R) ||| R) && \text{L2} \end{aligned}$$

$$\begin{aligned} \text{De asemenea } ((b \rightarrow R) ||| R) &= a \rightarrow ((b \rightarrow R) ||| (b \rightarrow R)) \square b \rightarrow (R ||| R) && \text{L6} \\ &= (a \rightarrow (b \rightarrow ((b \rightarrow R) ||| R))) && \text{ca mai} \\ &\quad \square b \rightarrow (a \rightarrow ((b \rightarrow R) ||| R)) && \text{sus} \\ &= \mu X. (a \rightarrow b \rightarrow X \\ &\quad \square b \rightarrow a \rightarrow X) \text{ deoarece recursivitatea este cu gardă.} \end{aligned}$$

Astfel $(R ||| R)$ este identic cu ex. 3.5 X2. O demonstrație similară ne duce la concluzia că $(AVS ||| AVS) = AVS2$. \square

3.6.2. Urme și refuzuri

O urmă a lui $(P ||| Q)$ este o întrefesere arbitrară a unei urme din P cu o urmă din Q . Pentru o definiție a întrefeserii a se vedea paragraful 1.9.3.

L1 $urme(P ||| Q) = \{s \mid \exists t \in urme(P). \exists u \in urme(Q). s \text{ întrefese } (t, u)\}$

$(P|||Q)$ se poate angaja în orice acțiune inițială posibilă atât pentru P cât și pentru Q și de aceea poate refuza numai acele mulțimi care sunt refuzate atât de P cât și de Q

$$L2 \quad \text{refuzuri}(P|||Q) = \text{refuzuri}(P \square Q)$$

Comportarea lui $(P|||Q)$ după angajarea în evenimentele urmei s este definită de formula cam elaborată

$$L3 \quad (P|||Q) \gamma s = \prod_{(t,u) \in T} (P \uparrow t) ||| (Q \uparrow u)$$

$$\text{unde } T = \{(t,u) \mid t \in \text{urme}(P) \wedge u \in \text{urme}(Q) \wedge s \text{ întreșese } (t,u)\}$$

Această lege reflectă faptul că nu există metodă de a afla în ce fel o urmă s a lui $(P|||Q)$ a fost construită ca o întreșesere a unei urme a lui P și a uneia a lui Q . Astfel că după s , comportarea ulterioară a lui $(P|||Q)$ poate reflecta orice întreșesere posibilă. Alegerea momentelor când se face întreșeserea nu este știută și nici determinată.

3.7 Specificații

În paragraful 3.4 am văzut necesitatea de a introduce mulțimile de refuzuri ca unul din aspectele observabile indirect ale comportării unui proces. În specificarea unui proces avem nevoie prin urmare de descrierea proprietăților dorite ale mulțimilor sale de refuzuri, la fel ca și pentru urme. Să utilizăm variabila *ref* pentru o mulțime de refuz arbitrară a unui proces la fel cum am folosit *ur* pentru a nota urma unui proces. Ca un rezultat, când P este un proces nedeterminist înțelesul său

$$P \text{ sat } S(ur, ref)$$

$$\text{este } \forall ur, ref. ur \in \text{urme}(P) \wedge ref \in \text{refuzuri}(P \uparrow ur) \Rightarrow S(ur, ref)$$

Exemple

X1 Când un *AV* a primit mai multe monezi decât ciocolate poate furniza, utilizatorul specifică că nu trebuie să refuze eliberarea unei ciocolate

$$ECHIT = (ur \downarrow choc < ur \downarrow mon \Rightarrow choc \notin ref)$$

Este de la sine înțeles că *orice* urmă *ur* și *orice* refuz *ref* al unui proces specificat, totdeauna va satisface această specificație. \square

X2 Când un *AV* a dat tot atâtea ciocolate câți bani s-au plătit, proprietarul specifică că nu trebuie să se refuze următoarea monedă

$$PROFIT1 = (ur \downarrow choc = ur \downarrow mon \Rightarrow mon \notin ref)$$

 \square

X3 Un *AVS* ar trebui să satisfacă ambele specificații

$$SPECNOI = ECHIT \wedge PROFIT1 \wedge (ur \downarrow choc \leq ur \downarrow mon)$$

Această specificație este satisfăcută de *AVS*. Este de asemenea satisfăcută de *AVS2* (1.1.3 X6) care va accepta câteva monezi la rând și apoi va da câteva ciocolate. \square

X4 Dacă se dorește, se poate limita numărul de monezi ce pot fi acceptate una după alta

$$CELMULT2 = (ur \downarrow mon - ur \downarrow choc \leq 2)$$

 \square

X5 Dacă se dorește, automatul poate accepta cel puțin două monezi odată ori de câte ori cumpărătorul oferă

$$CELPUTIN2 = (ur \downarrow mon - ur \downarrow choc < 2 \Rightarrow mon \notin ref)$$

 \square

X6 Procesul *STOP* refuză orice eveniment din alfabetul său. Următorul predicat specifică cum un proces cu alfabetul *A* nu se va opri niciodată

$$NONSTOP = (ref \neq A)$$

Dacă *P* sat *NONSTOP* și dacă mediul permite angajarea în toate evenimentele din *A*, procesul *P* trebuie să se angajeze în unul din ele. Deoarece (vezi X3 de mai înainte)

$$SPECNOIAV \Rightarrow ref \neq \{mon, choc\}$$

rezultă că orice proces care satisface *SPECNOIAV* nu se va opri. \square

Aceste exemple ne arată cum introducerea lui *ref* în specificarea unui proces permite exprimarea unui număr de proprietăți subtile și importante. Poate cea mai importantă dintre ele este aceea că un proces nu trebuie să se

oprească (X6). Aceste avantaje sunt obținute cu prețul unei complexități mărite în demonstrații și reguli.

De asemenea se dorește să se dovedească că un proces nu diverge. Paragraful 3.8 descrie cum un proces divergent poate face orice și refuza orice. Astfel, dacă există o mulțime care *nu poate* fi refuzată, procesul nu este divergent. Aceasta justifică formularea unei condiții suficiente pentru nondivergență.

$$NONDIV = (ref \neq A)$$

Din fericire

$$NONSTOP \equiv NONDIV$$

astfel că demonstrarea absenței divergenței nu presupune mai multă muncă decât demonstrarea absenței blocajului.

3.7.1 Reguli

În următoarele reguli de inferență o specificație va fi scrisă în oricare din formele S , $S(ur)$, $S(ur, ref)$ potrivit necesităților. În toate cazurile se va înțelege că specificația va conține ur și ref printre variabilele sale libere.

După definiția nedeterminismului, $(P \sqcap Q)$ se comportă ca P sau ca Q . De aceea, fiecare observație a comportării sale va fi o observație pentru P sau pentru Q sau pentru amândouă. De aceea această observație va fi descrisă prin specificarea lui P sau a lui Q sau a amândurora. Mai mult, inferența pentru nedeterminism are o formă excepțional de simplă.

L1 Dacă P sat S
și Q sat T
atunci $(P \sqcap Q)$ sat $(S \vee T)$

Legea pentru $STOP$ afirmă că acest proces nu face nimic și refuză orice.

L2A $STOP_A$ sat $(ur = \diamond \wedge ref \subseteq A)$

Deoarece refuzurile sunt totdeauna conținute în alfabet (3.4.1 L8) clauza $ref \subseteq A$ poate fi omisă. Astfel, dacă omitem alfabetele cu totul (așa cum o vom face în viitor), legea L2A este identică cu aceea pentru procese deterministe (1.10.2 L4A)

$STOP$ sat $ur = \diamond$

Legea anterioară pentru prefix (1.10.2 L4B) este de asemenea validă dar nu este suficient de puternică pentru a dovedi că procesul nu se poate opri înaintea unei acțiuni inițiale. Regula trebuie întărită prin menționarea faptului că în starea inițială, când $ur = \diamond$, acțiunea inițială nu poate fi refuzată.

L2B Dacă $P \text{ sat } S(ur)$

atunci $(c \rightarrow P) \text{ sat } ((ur = \diamond \wedge c \notin \text{ref}) \vee (ur_0 = c \wedge S(ur')))$

Legea pentru alegerea generală (1.10.2 L4) trebuie de asemenea întărită

L2 Dacă $\forall x \in B. P(x) \text{ sat } S(ur, x)$

atunci $(x.B \rightarrow P(x)) \text{ sat } ((ur = \diamond \wedge (B \cap \text{ref} = \{\})) \vee (ur_0 \in B \wedge S(ur', ur_0)))$

Legea pentru compunerea paralelă dată de 2.7 L1 este încă validă în condițiile în care specificațiile nu fac referire la mulțimi de refuzuri. Pentru a putea manipula corect refuzurile avem o lege ceva mai complicată.

L3 Dacă $P \text{ sat } S(ur, \text{ref})$

și $Q \text{ sat } T(ur, \text{ref})$

și nici P și nici Q nu diverg

atunci $(P \parallel Q) \text{ sat } (\exists X, Y. \text{ref} = (X \cup Y) \wedge S(ur \upharpoonright \alpha P, X) \wedge T(ur \upharpoonright \alpha Q, Y))$

Legea pentru schimbare de simbol necesită o modificare analoagă

L4 Dacă $P \text{ sat } S(ur, \text{ref})$

atunci $f(P) \text{ sat } S(f^{-1*}(ur), f^{-1}(\text{ref}))$ în condiția că f este injectivă

Legea pentru \square este surprinzător de simplă

L5 Dacă $P \text{ sat } S$

și $Q \text{ sat } T$

și nici P și nici Q nu diverg

atunci $(P \square Q) \text{ sat } (\text{if } ur = \diamond \text{ then } (S \wedge T) \text{ else } (S \vee T))$

Inițial, când $ur = \diamond$ o mulțime este refuzată de $(P \square Q)$ numai dacă este refuzată atât de P cât și de Q . De aceea această mulțime trebuie să fie descrisă de *ambele* specificații. Mai mult, când $ur \neq \diamond$, fiecare observație a lui $(P \square Q)$ trebuie să fie o observație atât a lui P cât și a lui Q și de aceea trebuie să fie descrisă de una din specificațiile lor (sau de ambele).

Legea pentru întretesere nu necesită să menționăm mulțimile de refuzuri.

- L6** Dacă $P \text{ sat } S(ur)$
 și $Q \text{ sat } T(ur)$
 și nici P nici Q nu diverg
 atunci $(P||Q) \text{ sat } (\exists s,t.(ur \text{ întrefese } (s,t) \wedge S(s) \wedge T(t)))$

Legea pentru mascare este mai complicată din necesitatea de a ne asigura că avem gardă în cazul divergenței

- L7** Dacă $P \text{ sat } (NODIV \wedge S(ur, ref))$
 atunci $(P \setminus C) \text{ sat } \exists s. ur = s \upharpoonright (\alpha P - C) \wedge S(s, ref \cup C)$

unde $NODIV$ afirmă că numărul simbolurilor mascate ce pot să apară este mărginit de o funcție a simbolurilor nemascate ce au apărut

$$NODIV = \#(ur \upharpoonright C) \leq f(ur \upharpoonright (\alpha P - C))$$

unde f este o funcție definită pe mulțimea urmelor cu valori în numere naturale.

Clauza $ref \cup C$ din consecința legii L7 necesită explicații. Ea provine din faptul că $P \setminus C$ poate refuza o mulțime X numai când P poate refuza întreaga mulțime $X \cup C$, cu alte cuvinte X împreună cu *toate* evenimentele mascate. $P \setminus C$ nu poate refuza să interacționeze cu mediul său extern până ce n-a ajuns la o stare în care să refuze angajarea în orice viitoare activitate internă mascată. Acest fel de echitate este o caracteristică importantă a oricărei definiții de mascare așa cum s-a descris în paragraful 3.5.1.

Regula pentru recursivitate (1.10.2 L6) trebuie de asemenea întărită. Fie $S(n)$ un predicat conținând variabila n din gama numerelor naturale $0, 1, 2, 3, \dots$

- L8** Dacă $S(0)$
 și $(X \text{ sat } S(n)) \Rightarrow (F(X) \text{ sat } S(n+1))$
 atunci $(\mu X. F(X)) \text{ sat } (\forall n. S(n))$

Această lege este bună chiar și pentru o recursivitate fără gardă cu toate că cea mai tare specificare ce poate fi dovedită pentru un astfel de proces este *true*.

3.8 Divergența

În capitolele anterioare am observat restricția ca ecuațiile care definesc un proces prin recursivitate să fie *cu gardă* (paragraful 1.1.2). Aceasta ne asigură

că ecuațiile au numai o singură soluție (1.3 L2). De asemenea restricția ne-a eliberat de a da o semnificație recursivității infinite

$$\mu X.X$$

Din nefericire, introducerea mascării (paragraful 3.5) înseamnă că o recursivitate aparent cu gardă poate să nu fie constructivă. De exemplu să considerăm ecuația

$$X = c \rightarrow X \setminus \{c\} + \{c\}$$

Aceasta are ca soluții atât $(c \rightarrow STOP)$ cât și $(c \rightarrow a \rightarrow STOP)$, care pot fi dovedite prin substituție.

Mai mult, orice ecuație recursivă care implică operatorul de mascare aplicat recursivității este potențial *fără gardă* și pasibilă de a avea mai mult de o soluție. Care soluție va fi cea corectă? Afirmăm că soluția corectă va fi cea mai puțin deterministă din cauză că aceasta permite o alegere nedeterministă între celelalte soluții. Cu această explicație putem scoate cu totul restricția ca recursivitățile să fie cu gardă și putem da o semnificație (posibil nedeterministă) *oricărei* expresii de forma $\mu X.F(X)$ unde F este definită în termenii oricărui operator introdus în această carte (cu excepția lui \setminus) și respectând toate constrângerile de alfabet.

Pentru a explica această semnificație vom trata întâi cel mai simplu caz, care (cum se întâmplă adesea) este cel mai defavorabil, recursivitatea infinită

$$\mu X.X$$

Orice proces este o soluție a ecuației recursive

$$X = X$$

Prin urmare, $\mu X.X$ se poate comporta în definitiv ca oricare proces. Este cel mai nedeterminist din toate procesele, cel mai puțin previzibil, cel mai puțin controlabil, pe scurt cel mai rău. Să-i dăm un nume potrivit și să-l definim

$$CHAOS_A = \mu X.A X$$

Un caz ceva mai bun este recursivitatea

$$\mu X.(c \rightarrow (X \setminus \{c\})) + \{c\} = c \rightarrow CHAOS$$

Acesta este un proces diferit de $CHAOS$ din cauză că se poate angaja cel puțin în evenimentul inițial c , înainte de a intra în $CHAOS$.

În afară de semnificația recursivității infinite, *CHAOS* este de asemenea rezultatul angajării unui proces într-o succesiune infinită de evenimente consecutive ascunse. Cel mai simplu și cel mai rău caz este un proces imediat divergent menționat anterior la sfârșitul paragrafului 3.5.1

$$\begin{aligned}
 (\mu X: A.(c \rightarrow X)) \setminus \{c\} &= \mu X: (A - \{c\}).(c \rightarrow X) \setminus \{c\} \\
 &= \mu X: A - \{c\}.(X \setminus \{c\}) && \text{din 3.5.1.L5} \\
 &= \mu X: A - \{c\}.X \\
 &= \text{CHAOS}_{A - \{c\}} && \text{definiția CHAOS}
 \end{aligned}$$

3.8.1 Legi

Deoarece *CHAOS* este cel mai nedeterminist proces nu poate fi schimbat prin aplicarea alegerii nedeterministe fiind de aceea un element neutru pentru \sqcap

$$\text{L1 } P \sqcap \text{CHAOS} = \text{CHAOS}$$

O funcție de procese care produce *CHAOS* când unul din argumentele sale este *CHAOS* se spune că este *strictă*. Legea de mai sus (plus simetria) afirmă că \sqcap este strictă. *CHAOS* este un astfel de proces îngrozitor încât orice proces care este definit în termeni de *CHAOS* este el însuși egal cu *CHAOS*

$$\text{L2 } \text{Următoarele operații sunt stricte}$$

$$\wedge, \parallel, f, \square, \setminus C, \parallel \text{ și } \mu X$$

Totuși prefixul nu este strict.

$$\text{L3 } \text{CHAOS} \neq (a \rightarrow \text{CHAOS})$$

din cauză că partea dreaptă se poate baza pe execuția lui a înainte de a deveni complet necontrolabilă.

Așa cum s-a mai spus, *CHAOS* este cel mai neprevizibil și cel mai necontrolabil dintre procese. Poate face orice, poate refuza orice

$$\text{L4 } \text{urme}(\text{CHAOS}_A) = A^*$$

$$\text{L5 } \text{refuzuri}(\text{CHAOS}_A) = \text{toate submulțimile lui } A$$

3.8.2 Divergențe

O *divergență* a unui proces este prin definiție orice urmă a unui proces după parcurgerea căreia procesul se comportă haotic. Mulțimea tuturor divergențelor este definită

$$\text{divergențe}(P) = \{s \mid s \in \text{urme}(P) \wedge (P/s) = \text{CHAOS}_{\alpha P}\}$$

Rezultă imediat că

$$\text{L1 } \text{divergențe}(P) \subseteq \text{urme}(P)$$

Din cauză că A este strictă

$$\text{CHAOS}_A = \text{CHAOS}$$

Urmează că divergențele unui proces sunt închise la concatenare în sensul că

$$\text{L2 } s \in \text{divergențe}(P) \wedge t \in (\alpha P)^* \Rightarrow (s^{\wedge} t) \in \text{divergențe}(P)$$

Deoarece CHAOS_A poate refuza orice submulțime a alfabetului său A

$$\text{L3 } s \in \text{divergențe}(P) \wedge X \subseteq \alpha P \Rightarrow X \in \text{refuzuri}(P/s)$$

Cele trei legi date mai sus pun în evidență proprietățile generale de divergență ale oricărui proces. Următoarele legi arată cum divergențele proceselor compuse sunt determinate de divergențele și urmele componentelor lor. Mai întâi, procesul *STOP* nu diverge niciodată

$$\text{L4 } \text{divergențe}(\text{STOP}) = \{\diamond\}$$

La cealaltă extremă, orice urmă a lui *CHAOS* ne conduce la *CHAOS*

$$\text{L5 } \text{divergențe}(\text{CHAOS}_A) = A^*$$

Un proces definit prin posibilitatea alegerii nu diverge în primul său pas. Mai mult, divergențele sale sunt determinate de ce se întâmplă după primul său pas.

$$\text{L6 } \text{divergențe}(x.B \rightarrow P(x)) = \{\langle x \rangle^{\wedge} s \mid x \in B \wedge s \in \text{divergențe}(P(x))\}$$

Orice divergență a lui P este de asemenea o divergență a lui $(P \sqcap Q)$ și a lui $(P \sqcup Q)$

$$\begin{aligned} \text{L7 } \text{divergențe}(P \sqcap Q) &= \text{divergențe}(P \sqcup Q) \\ &= \text{divergențe}(P) \cup \text{divergențe}(Q) \end{aligned}$$

Deoarece operatorul \parallel este strict, o divergență a lui $(P \parallel Q)$ pornește cu o urmă a activității nedivergente atât a lui P cât și a lui Q care ne conduce la divergența atât a lui P cât și a lui Q (sau a ambelor).

$$\text{L8 } \text{divergențe}(P \parallel Q) = \{s \mid t \in (\alpha P \cup \alpha Q)^* \wedge ((s \upharpoonright \alpha P \in \text{divergențe}(P) \wedge s \upharpoonright \alpha Q \in \text{urme}(Q)) \vee (s \upharpoonright \alpha P \in \text{urme}(P) \wedge s \upharpoonright \alpha Q \in \text{divergențe}(Q)))\}$$

O explicație similară se aplică lui $\parallel\parallel$

$$\text{L9 } \text{divergențe}(P \parallel\parallel Q) = \{u \mid \exists s, t. u \text{ întrețese}(s, t) \wedge ((s \in \text{divergențe}(P) \wedge t \in \text{urme}(Q)) \vee (s \in \text{urme}(P) \wedge t \in \text{divergențe}(Q)))\}$$

Divergențele unui proces rezultate din mascare includ urme provenite din divergențele originale plus acelea rezultate din încercarea de a masca o secvență infinită de simboluri.

$$\text{L10 } \text{divergențe}(P \backslash Q) = \{(s \upharpoonright (\alpha P - C)) \upharpoonright t \mid t \in (\alpha P - C)^* \wedge (s \in \text{divergențe}(P) \vee (\forall n. \exists u \in C^*. \#u > n \wedge (s \wedge u) \in \text{urme}(P)))\}$$

Un proces definit prin schimbare de simbol diverge numai dacă argumentul său diverge.

$$\text{L11 } \text{divergențe}(f(P)) = \{f^*(s) \mid s \in \text{divergențe}(P)\} \quad \text{în condițiile când } f \text{ este injectivă}$$

Este totuși neplăcut de a dedica atât de multă atenție divergenței când ea este ceva ce *nu* dorim să apară. Din nefericire, ea pare a fi o consecință inevitabilă a oricărei metode eficiente sau chiar procedurale de implementare. Divergența poate proveni din mascare sau recursivitate fără gardă. De aceea o parte a muncii unui proiectant de sistem constă în a dovedi că în proiectul său divergența nu va apare. Astfel pentru a dovedi că ceva nu poate să apară avem nevoie de o teorie matematică în care acel ceva să fie posibil.

3.9. Teoria matematică a proceselor nedeterminate

Legile date în acest capitol sunt evident mai complicate decât cele din capitolele anterioare. Justificările explicative și exemplele sunt prin urmare mai puțin convingătoare. Este de aceea important să construim o definiție mate-

matică adecvată a conceptului de proces nedeterminist și să dovedim corectitudinea legilor din definițiile operatorilor.

Ca și în paragraful 2.8.1, un model matematic se bazează pe proprietățile direct sau indirect relevante ale unui proces. Acestea includ desigur alfabetul și urmele sale, dar pentru un proces nedeterminist există de asemenea refuzurile sale (paragraful 3.4) și divergențele (paragraful 3.8). În plus față de refuzurile din primul pas al unui proces P , este necesar de a se lua totodată în calcul ce poate refuza P după angajarea într-o urmă arbitrară s a comportării sale. De aceea definim ca *eșecuri* ale unui proces P o relație (mulțime de perechi)

$$eșecuri(P) = \{(s, X) \mid s \in urme(P) \wedge X \in refuzuri(P/s)\}$$

Dacă (s, X) este un eșec a lui P , aceasta înseamnă că P se poate angaja în orice secvență de evenimente înregistrată prin s și apoi să refuze orice în ciuda faptului că mediul este pregătit să se angajeze în oricare din evenimentele lui X . Eșecurile unui proces informează mai mult asupra comportării procesului decât urmele sau refuzurile sale care pot fi amândouă definite în termenii eșecurilor

$$\begin{aligned} urme(P) &= \{s \mid \exists X. (s, X) \in eșecuri(P)\} \\ &= \text{domeniu}(eșecuri(P)) \\ refuzuri(P) &= \{X \mid (\langle \rangle, X) \in eșecuri(P)\} \end{aligned}$$

Felurile proprietăți ale urmelor (1.8.1 L6, L7, L8) și refuzurilor (3.4 L8, L9, L10, L11) pot fi ușor reformulate în termenii eșecurilor (vezi condițiile C0, C1, C2, C3 după definiția D0 de mai jos).

Acum suntem gata pentru decizia importantă ca un proces să fie definit de trei mulțimi specificând alfabetul, eșecurile și divergențele. Reciproc, orice trei mulțimi care satisfac anumite condiții definesc unic un proces. Vom defini mulțimea părților lui A ca mulțimea tuturor submulțimilor sale

$$P_A = \{X \mid X \subseteq A\}$$

D0 Un proces este un triplet (A, F, D)

unde A este orice mulțime de simboluri (pentru simplitate finită)

F este o relație între A^* și P_A

D este o submulțime a lui A^*

în cazul când satisfac următoarele condiții

$$\text{C0 } (\langle \rangle, \{\}) \in F$$

$$\text{C1 } (s \wedge t, X) \in F \Rightarrow (s, \{\}) \in F$$

$$C2 \quad (s, Y) \in F \wedge X \subseteq Y \Rightarrow (s, X) \in F$$

$$C3 \quad (s, X) \in F \wedge x \in A \Rightarrow (s, X \cup \{x\}) \in F \vee (s, X \setminus \{x\}) \in F$$

$$C4 \quad D \subseteq \text{domeniu}(F)$$

$$C5 \quad s \in D \wedge t \in A^* \Rightarrow s^{\wedge} t \in D$$

$$C6 \quad s \in D \wedge X \subseteq A \Rightarrow (s, X) \in F$$

(ultimele trei condiții reflectă legile 3.8.2 L1, L2, L3).

Cel mai simplu proces care satisface această definiție este și cel mai rău

$$D1 \quad CHAOS_A = (A, (A^* \times P_A), A^*)$$

unde $A^* \times P_A$ este produs cartezian

$$\{(s, X) \mid s \in A^* \wedge X \in P_A\}$$

Acest proces este cel mai mare proces cu alfabetul A , deoarece orice membru al lui A^* este atât o urmă cât și o divergență și orice submulțime a lui A este un refuz după toate urmele.

Alt proces simplu este definit

$$D2 \quad STOP_A = (A, \{\diamond\} \times P_A, \{\})$$

Acest proces nu face nimic, poate refuza orice și nu are divergențe.

Vom defini operatori pe mulțimea proceselor arătând cum cele trei mulțimi ale rezultatului provin din cele ale operanzilor. Desigur este necesar de arătat că rezultatul operației satisface cele șase condiții din D0. Aceasta se bazează pe presupunerea că operanzii permit acest lucru.

Cea mai simplă operație de definit este *sau* nedeterminist (\sqcap). Ca mulți alți operatori el este definit numai pentru operanzi cu același alfabet.

$$D3 \quad (A, F1, D1) \sqcap (A, F2, D2) = (A, F1 \cup F2, D1 \cup D2)$$

Procesul rezultat poate eșua sau diverge în toate cazurile în care unul din cei doi operanzi poate face aceasta. Legile 3.2.1 L1, L2, L3 sunt consecințe directe ale acestei definiții. Definițiile pentru ceilalți operatori pot fi date similar dar apare mai elegant de a scrie definiții separate pentru alfabet, eșecuri și divergențe. Definițiile pentru divergențe au fost date în paragraful 3.8.2, deci rămân de definit alfabetele și eșecurile.

$$D4 \quad \text{Dacă } \alpha P(x) = A$$

$$\text{și } B \subseteq A$$

$$\text{atunci } \alpha(x:B \rightarrow P(x)) = A$$

pentru toți x

- D5** $\alpha(P||Q) = (\alpha P \cup \alpha Q)$
D6 $\alpha(f(P)) = f(\alpha P)$
D7 $\alpha(P \square Q) = \alpha(P||Q) = \alpha P$ când $\alpha P = \alpha Q$
D8 $\alpha(P \setminus C) = \alpha P - C$
D9 $eşecuri(x:B \rightarrow P(x)) = \{ \Diamond, X \mid X \subseteq (\alpha P - B) \}$
 $\cup \{ \Diamond \wedge s, X \mid x \in B \wedge (s, X) \in eşecuri(P(x)) \}$
D10 $eşecuri(P||Q) = \{ s, (X \cup Y) \mid s \in (\alpha P \cup \alpha Q)^* \}$
 $\wedge (s \upharpoonright \alpha P, X) \in eşecuri(P)$
 $\wedge (s \upharpoonright \alpha Q, Y) \in eşecuri(Q)$
 $\cup \{ s, X \mid s \in divergenţe(P||Q) \}$
D11 $eşecuri(f(P)) = \{ f^*(s), f(X) \mid (s, X) \in eşecuri(P) \}$
D12 $eşecuri(P \square Q) = \{ s, X \mid (s, X) \in eşecuri(P) \cap eşecuri(Q) \}$
 $\cup \{ s \neq \Diamond \wedge (s, X) \in (eşecuri(P) \cup eşecuri(Q)) \}$
 $\cup \{ s, X \mid s \in divergenţe(P \square Q) \}$
D13 $eşecuri(P|||Q) = \{ s, X \mid \exists t, u. s \text{ întretrese } (t, u) \}$
 $\wedge (t, X) \in eşecuri(P)$
 $\wedge (u, X) \in eşecuri(Q)$
 $\cup \{ s, X \mid s \in divergenţe(P|||Q) \}$
D14 $eşecuri(P \setminus C) = \{ s \upharpoonright (\alpha P - C), X \mid (s, X \cup C) \in eşecuri(P) \}$
 $\cup \{ s, X \mid s \in divergenţe(P \setminus C) \}$

Explicarea acestor legi poate fi făcută funcție de urmele și refuzurile corespunzătoare împreună cu legile pentru /.

Rămâne de a da o definiție proceselor definite recursiv cu ajutorul lui μ . Metoda se bazează pe aceeași teoremă de punct fix ca în paragraful 2.8.2 cu excepția faptului că definiția ordinii este diferită.

$$\mathbf{D15} \quad (A, F1, D1) \sqsubseteq (A, F2, D2) \Leftrightarrow (F2 \subseteq F1 \wedge D2 \subseteq D1)$$

$P \sqsubseteq Q$ semnifică faptul că Q este egal cu P sau chiar mai mult în sensul divergenței și eşecului (mai puțin probabil să diveargă și să eşueze). Q este mai previzibil și mai controlabil decât P , din cauză că dacă Q poate face ceva nedorit, P poate face de asemenea, iar dacă Q poate refuza ceva, atunci P poate de asemenea. *CHAOS* poate face orice oricând și poate refuza orice oricând. Conform numelui, el este cel mai puțin previzibil și controlabil dintre procese sau pe scurt cel mai rău.

$$\mathbf{L1} \quad \text{CHAOS} \sqsubseteq P$$

Această ordonare este clar o ordine parțială. De fapt este o ordine parțială completă cu o operație de limită definită în termenii intersecțiilor lanțurilor descrescătoare de eșecuri și divergențe.

$$D16 \quad \bigcup_{n \geq 0} (A, F_n, D_n) = (A, \bigcap_{n \geq 0} F_n, \bigcap_{n \geq 0} D_n)$$

În condițiile când $(\forall n \geq 0. F_{n+1} \subseteq F_n \wedge D_{n+1} \subseteq D_n)$.

Operatorul μ este definit la fel ca pentru procese deterministe (2.8.2 L7) cu excepția unei diferențe în definiția ordinii care necesită ca *CHAOS* să fie folosit în locul lui *STOP*.

$$D17 \quad \mu X.A.F(X) = \bigcup_{i \geq 0} F^i(CHAOS_A)$$

Demonstrația că aceasta este soluția (de fapt cea mai nedeterministă soluție) unei ecuații date este aceeași cu cea dată în paragraful 2.8.2.

Ca și atunci, validitatea demonstrației depinde critic de faptul că toți operatorii folosiți în partea dreaptă a recursivității trebuie să fie continui conform ordinii. Din fericire, toți operatorii definiți în această carte (cu excepția lui /), sunt continui și astfel orice formulă construită cu ei. În cazul operatorului de mascare, continuitatea a fost una din cerințele principale pentru tratarea complexă a divergenței.

4 Comunicația

4.1 Introducere

În capitolele anterioare s-a introdus și ilustrat conceptul general de eveniment ca o acțiune fără durată a cărei apariție necesită participarea simultană a cel puțin două procese descrise independent. În acest capitol ne vom concentra asupra unei clase speciale de evenimente cunoscute ca evenimente de *comunicații*. O comunicație este un eveniment descris printr-o pereche

$c.v$

unde c este numele canalului pe care se face comunicația și v este valoarea mesajului transferat. Exemple ale acestei convenții au fost date deja în *COPIEBIT* (1.1.3 X7) și *LANȚ2* (2.6 X4).

Mulțimea tuturor mesajelor pe care P le poate transfera pe canalul c este definită

$$\alpha c(P) = \{v \mid c.v \in \alpha P\}$$

De asemenea definim funcțiile care extrag componenta de canal și mesaj dintr-o comunicație

$$\text{canal}(c.v) = c, \quad \text{mesaj}(c.v) = v$$

Toate operațiile introduse în acest capitol pot fi definite prin termenii conceptelor intuitive introduse în capitolele anterioare și multe din legi sunt cazuri particulare ale unor legi familiare anterioare. Motivul pentru introducerea notațiilor speciale constă în faptul că sunt sugestive în utilizare și în implementare. De asemenea impunerea unor restricții notaționale permite utilizarea unor metode de raționament mai puternice.

4.2 Intrare și ieșire

Fie v un membru din $\alpha c(P)$. Un proces care întâi transmite v pe canalul c și apoi se comportă ca P , este definit

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Singurul eveniment în care acest proces este pregătit inițial să se angajeze este evenimentul de comunicație $c.v$.

Un proces care este inițial pregătit să primească orice valoare x transferabilă pe canalul c , și apoi să se comporte ca $P(x)$ este definit

$$(c?x \rightarrow P(x)) = (y: \{y \mid \text{canal}(y) = c\} \rightarrow P(\text{mesaj}(y)))$$

Exemple

X0 Folosind noile definiții pentru intrare și ieșire putem rescrie 1.1.3 X7

$$\text{COPIEBIT} = \mu X. (in?x \rightarrow (out!x \rightarrow X))$$

unde $\alpha in(\text{COPIEBIT}) = \alpha out(\text{COPIEBIT}) = \{0,1\}$ □

Se observă convenția ca aceste canale să fie folosite pentru comunicații unidirecționale și numai între două procese. Un canal care este folosit numai pentru transferuri de ieșire într-un proces se va chema canal de ieșire (transmisie) a celui proces, iar unul folosit numai pentru transferuri de intrare, se va chema canal de intrare (recepție). În ambele cazuri, vom spune într-un sens mai larg că numele unui canal este un membru al alfabetului procesului.

Când desenăm o diagramă de conexiune (paragraful 2.4) a unui proces, canalele sunt desenate cu săgeți în direcția corespunzătoare sensului de transfer și etichetate cu numele canalului (fig. 4.1).

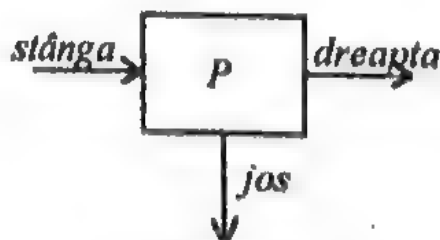


Figura 4.1

Fie P și Q procese și c un canal de ieșire pentru P și intrare pentru Q . Când P și Q sunt compuse concurrent într-un sistem $(P||Q)$, comunicația va

apare pe canalul c de fiecare dată când P emite un mesaj și Q simultan recepționează acel mesaj. Un proces care emite un mesaj precizează (specifică unic) valoarea lui, iar un proces care recepționează este pregătit să accepte orice valoare comunicabilă. Evenimentul care va apare de fapt este comunicația $c.v$ unde v este valoarea specificată în procesul care transmite. Aceasta necesită constrângerea evidentă ca acel canal c să aibă același alfabet la ambele capete

$$\alpha c(P) = \alpha c(Q)$$

Pe viitor vom presupune îndeplinită această constrângere și unde nu se poate naște nici o confuzie vom scrie αc pentru $\alpha c(P)$. Un exemplu pentru funcționarea acestui model de comunicație a fost dat în *LANT2* (2.6 X4). Exemple și mai interesante vor fi date în paragraful 4.3 și următoarele.

În general, mesajul ce va fi transmis de un proces este specificat cu ajutorul unei expresii conținând variabile cărora li s-a asignat o valoare într-o recepție anterioară, așa cum se arată în exemplele următoare.

Exemple

X1 Un proces care copie imediat orice mesaj pe care îl recepționează pe canalul din stânga, transmițându-l prin canalul din dreapta

$$\begin{aligned}\alpha st\acute{a}nga(COPIE) &= \alpha dreapta(COPIE) \\ COPIE &= \mu X.(st\acute{a}nga?x \rightarrow dreapta!x \rightarrow \lambda)\end{aligned}$$

Dacă $\alpha st\acute{a}nga = \{0,1\}$, *COPIE* este aproape identic cu *COPIEBIT* (1.1.3 X7) □

X2 Un proces asemănător lui *COPIE* dar care dublează valoarea la ieșire

$$\begin{aligned}\alpha st\acute{a}nga &= \alpha dreapta = \mathbb{N} \\ DUBLU &= \mu X.(st\acute{a}nga?x \rightarrow dreapta!(x+x) \rightarrow \lambda)\end{aligned}$$
□

X3 O cartelă este o secvență de 80 de caractere care pot fi citite ca un singur mesaj prin canalul *stâng* al unui proces. Un proces care citește cartele și transmite caracterele lor unul câte unul

$$\alpha st\acute{a}nga = \{s \mid s \in \alpha dreapta^* \wedge \#s = 80\}$$

$$DESPACHET = P_{\diamond}$$

unde $P_{\diamond} = st\acute{a}nga?s \rightarrow P_s$,

și $P_{\infty} = dreapta!x \rightarrow P_{\diamond}$

$P_{<x>\wedge s} = dreapta!x \rightarrow P_s$

□

X4 Un proces recepționează caracterele prin canalul din stânga unul câte unul și le assemblează în linii de câte 125 caractere lungime. Fiecare linie completă este transmisă apoi prin canalul din dreapta ca un mesaj-vector

$\alpha dreapta = \{s \mid s \in \alpha st\acute{a}nga^* \wedge \#s = 125\}$

$\hat{IMPACHET} = P_{\diamond}$

dacă $\#s = 125$

unde $P_s = dreapta!s \rightarrow P_{\diamond}$

dacă $\#s < 125$

și $P_s = st\acute{a}nga?x \rightarrow P_s \wedge \infty$

Aici P_s descrie comportarea unui proces care recepționează și împachetează caracterele în secvența s , ele urmând să fie transmise când linia este completă. □

X5 Un proces care tot ce recepționează prin canalul din stânga copie transmițând prin canalul din dreapta, exceptând perechile de asteriscuri consecutive înlocuite cu un singur semn " \uparrow "

$\alpha st\acute{a}nga = \alpha dreapta - \{\uparrow\}$

$ELIMIN = \mu X. st\acute{a}nga?x \rightarrow$

if $x \neq "*" \text{ then } (dreapta!x \rightarrow X)$

else $st\acute{a}nga?y \rightarrow (\text{if } y = "*" \text{ then } (dreapta!\uparrow \rightarrow X)$

else $(dreapta!"*" \rightarrow dreapta!y \rightarrow X))$ □

Un proces trebuie să fie pregătit inițial să comunice pe oricare dintr-o mulțime de canale, alegerea acestora lăsând-o altor procese cu care este conectat. Pentru aceasta, adaptăm notația alegerii introdusă în cap. 1. Dacă c și d sunt nume de canale distincte

$(c?x \rightarrow P(x) \mid d?y \rightarrow Q(y))$

semnifică un proces care inițial recepționează x pe c și apoi se comportă ca $P(x)$ sau inițial recepționează y pe canalul d și apoi se comportă ca $Q(y)$. Alegerea este determinată de oricare dintre ieșirile corespunzătoare care este gata, așa cum se arată mai jos.

Deoarece am decis să facem abstracție de timingul evenimentelor și de viteza proceselor care se angajează în ele, ultima exprimare din paragraful precedent necesită explicații. Considerăm cazul când canalele c și d sunt ca-

nale de ieșire ale altor două procese separate care sunt independente în sensul că nu comunică direct sau indirect între ele. Acțiunile acestor două procese sunt de aceea întreșesute. Astfel, dacă un proces progresează în sensul transmiterii unui mesaj pe canalul c și celălalt progresează în sensul transmiterii unui mesaj pe d , nu este determinat care dintre ele va face primul transmisia. Un implementator va dori, într-un fel sau altul, să rezolve acest nedeterminism de exemplu în favoarea primei ieșiri ce devine disponibilă. Această politică protejează de asemenea împotriva blocajului care rezultă dacă a doua ieșire nu va apare niciodată sau dacă apare numai după prima, ca în cazul când ambele canale c și d aparțin aceluiași proces concurent care transmite pe unul și apoi pe celălalt canal

$$(c!2 \rightarrow d!4 \rightarrow P)$$

De aceea prezența alternativei intrărilor nu numai că protejează împotriva blocajului dar de asemenea realizează o eficiență sporită și reduce timpii de acces la comunicațiile propuse. Un călător care așteaptă autobuzul 28 de exemplu, va trebui să aștepte mai mult decât cel care este pregătit să călătorească și cu autobuzul 9, și cu 28, oricare ar sosi primul în stație. Având în vedere sosirile aleatoare, călătorul care are cele două alternative va aștepta, paradoxal, jumătate din timp. Astfel, a aștepta primul dintre mai multe evenimente posibile este singura cale de a realiza dezideratul de mai sus. A cumpăra computere mai rapide este inutil.

X6 Un proces care recepționează pe oricare două canale *stânga1* sau *stânga2* și imediat transmite mesajul în dreapta

$$\begin{aligned} \alpha \text{stânga1} &= \alpha \text{stânga2} = \alpha \text{dreapta} \\ \text{UNIFICĂ} &= (\text{stânga1?}x \rightarrow \text{dreapta!}x \rightarrow \text{UNIFICĂ} \\ &\quad | \text{stânga2?}x \rightarrow \text{dreapta!}x \rightarrow \text{UNIFICĂ}) \end{aligned}$$

Ieșirea acestui proces este întreșeserea mesajelor de intrare de la canalele *stânga1* și *stânga2*. □

X7 Un proces care totdeauna este pregătit să primească o valoare prin *stânga* sau să transmită în dreapta valoarea pe care tocmai a recepționat-o

$$\begin{aligned} \alpha \text{stânga} &= \alpha \text{dreapta} \\ \text{VAR} &= \text{stânga?}x \rightarrow \text{VAR}_x \end{aligned}$$

$$\text{unde } \text{VAR}_x = (\text{stânga?}y \rightarrow \text{VAR}_y \\ | \text{dreapta?}x \rightarrow \text{VAR}_x)$$

Aici procesul VAR_x se comportă ca o variabilă de program cu valoarea curentă x . Valori noi îi sunt asignate prin comunicațiile primite de la canalul său stâng și valoarea sa curentă este accesată prin comunicarea făcută prin canalul din dreapta. Dacă $\alpha_{st\grave{a}nga} = \{0,1\}$, comportarea lui VAR este aproape identică cu cea a lui $BOOL$ (2.6 X5). \square

X8 Un proces care primește de la canalele *sus* și *stânga* și emite prin canalul *jos* o funcție de intrări înainte de a repeta acțiunea

$$NOD(v) = \mu X. (sus?sum\grave{a} \rightarrow st\grave{a}nga?prod \rightarrow \\ jos!(sum\grave{a} + v \times prod) \rightarrow X)$$

 \square

X9 Un proces care este totdeauna gata să recepționeze un mesaj prin stânga și să transmită prin dreapta primul mesaj primit dar netransmis încă

$$BUFFER = P_{\diamond}$$

$$\text{unde } P_{\diamond} = st\grave{a}nga?x \rightarrow P_{\infty}$$

$$\text{și } P_{\infty} \wedge_s = (st\grave{a}nga?y \rightarrow P_{\infty} \wedge_s \wedge_{\diamond} | \\ dreapta!x \rightarrow P_s)$$

$BUFFER$ se comportă ca o coadă. Mesajele se alătură capătului din stânga al cozii și-o părăsesc prin partea dreaptă în aceeași ordine în care s-au alăturat dar după o întârziere posibilă, timp în care alte mesaje se pot alătura cozii. \square

X10 Un proces se comportă ca o stivă de mesaje. Când este goală, răspunde cu semnalul *gol*. Totdeauna este gata de a primi un nou mesaj prin canalul din stânga și de a-l pune pe stivă. Ori de câte ori nu este goală, procesul stivă este gata de a transmite prin canalul din dreapta și a înlocui vârful său

$$STIV\grave{A} = P_{\diamond}$$

$$\text{unde } P_{\diamond} = (gol \rightarrow P_{\diamond} | st\grave{a}nga?x \rightarrow P_{\infty})$$

$$\text{și } P_{\infty} \wedge_s = (dreapta!x \rightarrow P_s | st\grave{a}nga?y \rightarrow P_{\infty} \wedge_s \wedge_{\diamond})$$

Acest proces este foarte asemănător cu procesul precedent, cu excepția faptului că atunci când stiva este goală procesul aferent participă la evenimentul *gol* și de asemenea noile mesaje sosite sunt adăugate la același capăt al secvenței depuse din care se și ia când este cazul. Astfel, dacă y este noul mesaj de in-

trare și x este mesajul curent gata pentru transmis, procesul *STIVĂ* reține $\langle y \rangle \wedge \langle x \rangle \wedge s$ pe când *BUFFER* reține $\langle x \rangle \wedge s \wedge \langle y \rangle$. \square

4.2.1 Implementare

Într-o implementare LISP a proceselor comunicante evenimentul $c.v$ este natural reprezentat de perechea cu punct $(c.v)$ și este construită cu

$cons("c,v).$

Comenzile de recepție și de transmisie sunt convenabil implementate ca funcții care întâi preiau numele canalului ca argument. Dacă procesul nu este pregătit să comunice pe canal, dă răspunsul "BLIP. Valoarea efectivă transmisă prin comunicație este tratată separat în următorul pas, cum se arată mai jos.

Dacă Q este comanda de intrare

$(c?x \rightarrow Q(x))$

atunci $Q("c) \neq \text{"BLIP"}$. În afară de aceasta, rezultatul ei este o funcție care așteaptă valoarea de intrare x ca argument și generează ca rezultat procesul $Q(x)$. Astfel Q este implementat prin apelul funcției LISP

$recep("c, \lambda x.Q(x))$

care este definită

$recep(c,F) = \lambda y. \text{ if } y \neq c \text{ then "BLIP" else } F$

Urmează că expresia $Q\langle c.v \rangle$ este reprezentat în LISP de $Q("c)(v)$ în condițiile când $\langle c.v \rangle$ este o urmă a lui Q .

Dacă P este comanda de ieșire

$(c!v \rightarrow P)$

atunci $P("c) \neq \text{"BLIP"}$. Altfel, rezultatul său este perechea $cons(v,P)$. Astfel, P este implementat prin apelul funcției LISP

$transmit("c,v,P)$

care este definită

$$\begin{array}{l}
 0 \\
 s \leq t \Leftrightarrow (s=t) \\
 \begin{array}{ccc} n & m & n+m \\ s \leq t \wedge t \leq u \Rightarrow s \leq u \\ \\ n \\ s \leq t \Leftrightarrow \exists n. s \leq t \end{array}
 \end{array}$$

Exemple

X1 COPIE sat dreapta \leq^1 stânga □

X2 DUBLU sat dreapta \leq^1 dublu \ast (stânga) □

X3 DESPACHET sat dreapta \leq^1 stânga

unde $\wedge \langle s_0, s_1, \dots, s_{n-1} \rangle \Rightarrow s_0 \wedge s_1 \wedge \dots \wedge s_{n-1}$

Aici specificarea obligă ca secvența transmisă prin partea dreaptă să fie obținută din concatenarea succesivă a secvențelor recepționate în stânga. □

X4 ÎMPACHET sat $((\wedge^{125} \text{dreapta} \leq \text{stânga}) \wedge (\# \ast \text{dreapta}) \in \{125\} \ast)$

Această specificare afirmă că orice element transmis în dreapta este el însuși o secvență de 125 caractere și concatenarea tuturor acestor secvențe este o sub-secvență a ceea ce s-a recepționat în stânga. □

Dacă \oplus este un operator binar este convenabil să-l aplicăm distributiv elementelor corespondente ale celor două secvențe. Lungimea secvenței rezultante este egală cu aceea a celui mai scurt operand

$$\begin{array}{ll}
 s \oplus t = \diamond & \text{dacă } s = \diamond \text{ sau } t = \diamond \\
 = (s_0 \oplus t_0) \wedge (s' \oplus t') & \text{altfel}
 \end{array}$$

Mai clar $(s \oplus t)[i] = s[i] \oplus t[i]$ pentru $i < \min(\#s, \#t)$

și $s \leq t \Rightarrow (s \oplus u \leq t \oplus u) \wedge (u \oplus s \leq u \oplus t)$

X5 Șirul lui Fibonacci

$$\langle 1, 1, 2, 3, 5, 8, \dots \rangle$$

este definit prin relația de recurență:

$$\begin{array}{l}
 fib[0] = fib[1] = 1 \\
 fib[i+2] = fib[i+1] + fib[i]
 \end{array}$$

A doua relație poate fi rescrisă folosind operatorul ' care deplasează secvența de numere o poziție la stânga

$$fib'' = fib' + fib$$

Definiția originală a șirului lui Fibonacci se poate rescrie într-o formă mai complicată folosind indexul

$$fib''[i] = (fib' + fib)[i]$$

$$\begin{array}{ll} \text{de aceea} & fib'[i+1] = fib'[i] + fib[i] \\ \text{și} & fib[i+2] = fib[i+1] + fib[i]. \end{array} \quad (1.9.4 L1)$$

Altă semnificație a ecuației se poate face printr-o descriere a sumei infinite, unde deplasarea spre stânga o poziție este arătată clar

$$\begin{array}{rcl} 1, 1, 2, 3, 5, \dots & fib \\ // // // & \\ 1, 2, 3, 5, \dots & + fib' \\ // // & \\ 2, 3, 5, \dots & = fib'' \end{array}$$

În situația de mai sus, fib este privită ca o secvență infinită. Dacă s este o secvență inițială finită a lui fib (cu $\#s \geq 2$), atunci în loc de ecuație avem inegalitatea

$$s' \leq s' + s$$

Această formulare poate fi folosită pentru a specifica un proces FIB care transmite secvența Fibonacci prin canalul din dreapta

$$FIB \text{ sat } (dreapta \leq \langle 1, 1 \rangle \vee (\langle 1, 1 \rangle \leq dreapta \wedge dreapta'' \leq dreapta' + dreapta)) \quad \square$$

X6 O variabilă cu valoarea x transmite în dreapta cea mai recentă valoare recepționată la intrare în stânga sau x dacă nu a fost nimic la intrare. Mai formal, dacă cea mai recentă acțiune a fost o transmisie, atunci valoarea care a fost transmisă este egală cu ultimul termen din secvența $\langle x \rangle^{\wedge stânga}$

$$VAR_x \text{ sat } (\overline{canal(ur_0)} = \overline{dreapta} \Rightarrow \overline{dreapta_0} = (\langle x \rangle^{\wedge stânga})_0)$$

unde $\overline{s_0}$ este ultimul element al lui s (paragraful 1.9.5).

Acesta este un exemplu de proces care poate fi specificat adecvat numai în termenii unei secvențe de mesaje pe canale separate proprii. Este de asemenea necesar să știm ordinea comunicațiilor întrețesute pe canalele separate, de exemplu că ultima comunicație este prin canalul din dreapta. În general, această extra complexitate va fi necesară pentru procese care utilizează operatorul alegere. \square

X7 Procesul *UNIFICĂ* produce o întrețesere (paragraful 1.9.3) a două secvențe recepționate prin *stânga1* și *stânga2*, reținând numai cel mult un mesaj

$$UNIFICĂ \text{ sat } \exists r. \overset{1}{dreapta} \leq r \wedge r \text{ întrețese}(stânga1, stânga2) \quad \square$$

X8 *BUFFER* sat $dreapta \leq stânga$ \square

Un proces care satisface specificația ($dreapta \leq stânga$) descrie comportarea unui protocol de comunicație transparent care garantează livrarea prin dreapta numai a acelor mesaje care au sosit în stânga, în aceeași ordine. Un protocol realizează aceasta în ciuda faptului că locul unde se generează mesajele este complet separat de locul unde sunt ele recepționate și de faptul că mediul de comunicație care interconectează cele două contexte este oarecum nesigur. Exemple vor fi date în paragraful 4.4.5.

4.3 Comunicații

Fie P și Q procese și fie c un canal folosit la transmisie de P și la recepție de Q . Astfel mulțimea conținând toate evenimentele de comunicații de forma $c.v$ este conținută în intersecția alfabetului lui P cu alfabetul lui Q . Când aceste procese se compun concurent în sistemul $(P||Q)$ poate să apară o comunicație $c.v$ numai când ambele procese se angajează simultan în acel eveniment, cu alte cuvinte când P transmite o valoare v pe canalul c și Q simultan primește aceeași valoare. Un proces receptor este pregătit să accepte orice valoare comunicabilă astfel că este sarcina procesului emițător de a determina ce valoare-mesaj este transmisă cu fiecare ocazie, ca în 2.6 X4.

Transmisia poate fi astfel privită ca un caz special al operatorului prefix și recepția ca un caz special al alegerii, aceasta conducându-ne la legea

$$L1 \quad (c!v \rightarrow P) || (c?x \rightarrow Q(x)) = c!v \rightarrow (P || Q(v))$$

De remarcat că $c!v$ rămâne în dreapta ecuației ca o acțiune observabilă în comportarea sistemului. Aceasta reprezintă posibilitatea fizică de a intercepta firele ce conectează componentele sistemului și de aceea de a fi la curent cu comunicația lor internă. Este de asemenea de ajutor în raționamentele despre sistem. Dacă se dorește, astfel de operații interne pot fi eliminate prin operatorul de mascare descris în paragraful 3.5 aplicat în afara compunerii paralele a două procese care comunică pe același canal, cum se arată în legea

$$L2 \quad (c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)) \setminus C = (P \parallel Q(v)) \setminus C$$

unde $C = \{c.v \mid v \in \alpha c\}$

Exemple vor fi date în paragraful 4.4 și 4.5.

Specificarea compunerii paralele a proceselor comunicante ia o formă particulară simplă când numele canalelor sunt utilizate pentru a semnifica secvențele de mesaje ce trec de-a lungul lor. Fie c numele unui canal prin care P și Q comunică. În specificarea lui P , c semnifică secvența de mesaje comunicate de P prin c . Similar, în specificarea lui Q , c reprezintă secvența de mesaje comunicate de Q . Din fericire, prin natura însăși a comunicației, când P și Q comunică prin c , secvențele de mesaje transmise și recepționate trebuie tot timpul să fie identice. Deci, această secvență trebuie să satisfacă atât specificarea lui P cât și a lui Q . Aceasta este valabil pentru toate canalele aparținând intersecției alfabetelor lor.

Să considerăm acum un canal din alfabetul lui P dar *nu* și din cel al lui Q . Acest canal nu poate fi menționat în specificarea lui Q astfel că valorile transmise pe el sunt constrânse numai de specificarea lui P . Similar, Q este cel care determină proprietățile comunicației pe propriile sale canale. Prin urmare, o specificare a comportării lui $(P \parallel Q)$ poate fi simplu formată printr-o conjuncție logică între specificările lui P și a lui Q . Totuși, această simplificare este valabilă numai dacă specificările lui P și ale lui Q sunt exprimate în întregime în termenii numelor canalelor, care nu este totdeauna posibil așa cum se arată în 4.2.2 X6.

Exemple

$$X1 \quad \begin{aligned} P &= (st\acute{a}nga!x \rightarrow mij!(x \times x) \rightarrow P) \\ Q &= (mij!y \rightarrow dreapta!(173 \times y) \rightarrow Q) \end{aligned}$$

De fapt,

$$P \text{ sat } \overset{1}{mij \leq p\acute{a}trat * (st\acute{a}nga)}$$

$$\text{și } Q \text{ sat } \overset{1}{dreapta \leq 173 \times mij}$$

unde $(173 \times mij)$ multiplică fiecare mesaj al lui mij cu 173.

Urmează că

$$(P \parallel Q) \text{ sat } (dreapta \leq 173 \overset{1}{mij}) \wedge (mij \leq \overset{1}{pătrat} * (stânga))$$

Aici specificarea implică

$$dreapta \leq 173 \times pătrat * (stânga)$$

care a fost de fapt intenția originală. \square

Când procesele comunicante sunt conectate prin operatorul concurență \parallel , formulele ce rezultă sunt foarte sugestive pentru o implementare fizică în care componentele electronice sunt conectate prin fire prin care să comunice. Scopul unei astfel de implementări este de a mări viteza de producere a rezultatelor utile. Tehnica este mai ales efectivă când același calcul poate să fie realizat pe fiecare membru al unui flux de date de intrare și rezultatele trebuie să fie transmise cu aceeași viteză ca și recepția, posibil după o întârziere inițială. Astfel de sisteme se numesc *rețele de fluxuri de date* (*data flow networks*).

O reprezentare a unui sistem de procese comunicante se apropie mult de realizare lor fizică. Un canal de transmisie a unui proces este legat la un canal de recepție cu același nume al altui proces, lăsându-se libere doar canalele proprii din alfabetele proceselor. Astfel exemplul X1 poate fi desenat ca în fig 4.2.



Figura 4.2

- X2 Două fluxuri de numere trebuie recepționate prin canalele *stânga1* și *stânga2*. Pentru fiecare x recepționat prin *stânga1* și fiecare y prin *stânga2* numărul $(a \times x + b \times y)$ trebuie transmis prin dreapta. Necesitățile de viteză impun ca multiplicările să se facă concurrent. De aceea definim două procese și le compunem

$$X21 = (stânga1 ? x \rightarrow mij | (a \times x) \rightarrow X21)$$

$$X22 = (stânga2 ? y \rightarrow mij ? z \rightarrow dreapta | (z + b \times y) \rightarrow X22)$$

$$X2 = (X21 \parallel X22)$$

De fapt, $X2 \text{ sat } (mij \leq a \times \overset{1}{stânga1} \wedge dreapta \leq mij + b \times \overset{1}{stânga2})$
 $\Rightarrow (dreapta \leq a \times stânga1 + b \times stânga2)$ \square

X3 Un flux de numere este recepționat prin *stânga* iar în dreapta este transmisă o sumă ponderată a perechilor consecutive de numere introduse, cu ponderile *a* și *b*.

Mai precis, vrem ca

$$dreapta \leq a \times stânga + b \times stânga'$$

Soluția se poate contura prin adăugarea unui nou proces X23 la soluția lui X2

$$X3 = (X2 \parallel X23)$$

unde $X23 \text{ sat } (stânga1 \leq stânga \wedge stânga2 \leq stânga')$

X23 poate fi definit

$$X23 = (stânga?x \rightarrow stânga1!x \rightarrow (\mu X. stânga?x \rightarrow stânga2!x \rightarrow stânga1!x \rightarrow X))$$

Practic procesul copie valoarea recepționată prin *stânga* atât la *stânga1* cât și la *stânga2*, dar omite primul element în cazul lui *stânga2*.

O imagine a rețelei X3 este în fig. 4.3. □



Figura 4.3

Când două procese concurente comunică între ele prin transmitere și recepționare numai pe un singur canal, ele nu se pot bloca (compară 2.7 L2). Rezultă că orice rețea de procese ce funcționează continuu și este fără cicluri nu se poate bloca deoarece un graf fără cicluri poate fi descompus în subgrafi componente conectate numai printr-un singur arc. Totuși rețeaua din X3 conține un ciclu indirect iar rețelele ciclice nu pot fi descompuse în subrețele cu excepția conexiunilor pe două sau mai multe canale. Astfel, în acest caz, absența blocajului nu poate fi atât de ușor asigurată. De exemplu, dacă două ieșiri $stânga2!x \rightarrow stânga1!x \rightarrow$ din bucla lui X23 sunt inversate, blocajul ar putea apărea imediat. Pentru a dovedi absența blocajului este posibil de multe ori să ignorăm conținutul mesajelor și să privim fiecare comunicație pe canalul *c* ca un singur eveniment numit *c*. Comunicațiile pe canalele neconectate pot fi ignorate. Astfel X3 poate fi scris în termenii acestor evenimente

$$\begin{aligned}
&(\mu X.st\acute{a}nga1 \rightarrow mij \rightarrow X) \\
&\parallel (\mu Y.st\acute{a}nga2 \rightarrow mij \rightarrow Y) \\
&\parallel (st\acute{a}nga1 \rightarrow (\mu Z.st\acute{a}nga2 \rightarrow st\acute{a}nga1 \rightarrow Z)) \\
&= \mu X3.(st\acute{a}nga1 \rightarrow st\acute{a}nga2 \rightarrow mij \rightarrow X3)
\end{aligned}$$

Folosind metode algebrice ca în 2.3 X1, cele de mai sus dovedesc că X3 nu se poate bloca.

Aceste exemple ne arată cum pot fi construite rețele de fluxuri de date pentru a calcula unul sau mai multe fluxuri de rezultate din unul sau mai multe fluxuri de date de intrare. Forma rețelei corespunde îndeaproape structurii operanzilor și operatorilor ce apar în expresiile de calcul. Când aceste modele sunt mari, dar regulate, este convenabil să introducem nume subscrise pentru canale și notație iterativă pentru combinația concurentă

$$\parallel_{i < n} P(i) = (P(0) \parallel P(1) \parallel \dots \parallel P(n-1))$$

O rețea regulată de acest fel este cunoscută ca *masiv iterativ*. Dacă diagrama de conexiune nu are cicluri directe, se utilizează adesea termenul de *masiv sistolic*, deoarece datele trec prin sistem la fel ca sângele prin camerele inimii.

X4 Canalele $\{st\acute{a}nga_j \mid j < n\}$ sunt folosite pentru a prelua coordonatele unor puncte succesive într-un spațiu n -dimensional. Fiecare mulțime de coordonate trebuie multiplicată cu un vector fix V de lungime n și produsul scalar rezultat trebuie transmis în dreapta. Mai formal

$$dreapta \leq \sum_{j=0}^{n-1} V_j \times st\acute{a}nga_j$$

Se specifică că în fiecare microsecundă cele n coordonate ale unui punct trebuie recepționate și produsul scalar transmis. Viteza fiecărui procesor individual este astfel că îi trebuie aproape 1 μ s pentru o recepție, o multiplicare, o adunare și o transmisie. De aceea, este clar că este nevoie de cel puțin n procesoare care să lucreze concurrent. Soluția problemei ar putea fi realizată printr-un masiv iterativ cu cel puțin n elemente.

Să înlocuim în specificație operatorul Σ cu definiția lui inductivă

$$mij_0 = 0^*$$

$$mij_{j+1} = V_j \times st\acute{a}nga_j + mij_j,$$

$$dreapta = mij_n$$

pentru $j < n$

Astfel, am despărțit specificația într-o conjuncție de $n+1$ ecuații componente, fiecare conținând cel mult o multiplicare. Tot ce se cere este de a scrie un proces pentru fiecare ecuație

$$MULT_0 = (\mu X. mij_0!0 \rightarrow X)$$

$$MULT_{j+1} = (\mu X. st\acute{a}nga_j!x \rightarrow mij_j!y \rightarrow mij_{j+1}!(V_j x + y) \rightarrow X) \quad \text{pentru } j < n$$

$$MULT_{n+1} = (\mu X. mij_n!x \rightarrow dreapta!x \rightarrow X)$$

$$RE\acute{T}EA = \prod_{j=0}^{n+1} MULT_j$$

Diagrama de conexiune este dată în fig. 4.4. □

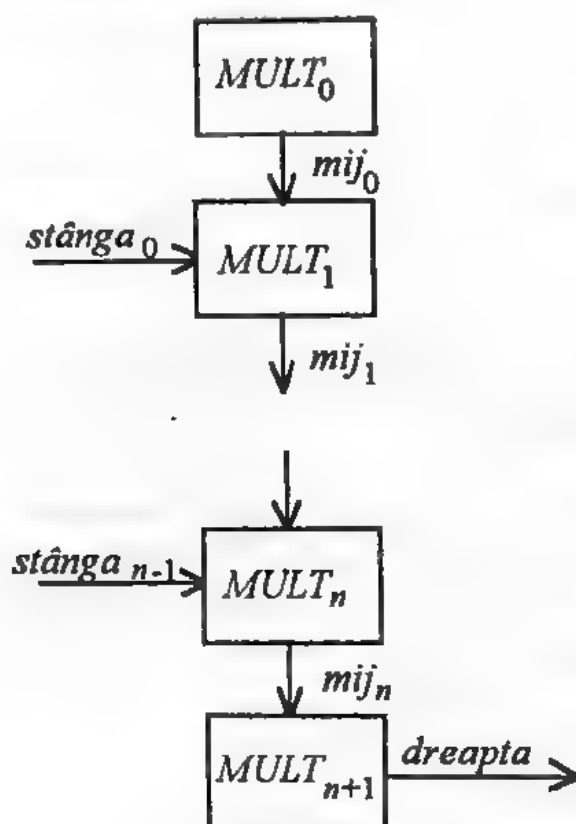


Figura 4.4

X5 Acest exemplu este similar cu X4 cu excepția faptului că cele m produse scalare diferite ale aceleiași mulțimi de coordonate sunt necesare aproape simultan. Practic, canalul $st\acute{a}nga_j$ (pentru $j < n$) trebuie folosit pentru recepția coloanei j a unui masiv infinit, aceasta la rândul ei trebuie multiplicată cu matricea $M(n \times m)$ și coloana i a rezultatului trebuie transmisă prin $dreapta_i$ pentru $i < m$. Ca formulă

$$dreapta_i = \sum_{j(n)} M_{ij} \times st\acute{a}nga_j$$

Coordonatele rezultatului sunt necesare mai rapid ca înainte astfel că sunt necesare cel puțin $m \times n$ procese.

Soluția poate găsi aplicație practică pentru un dispozitiv de afișare grafică care transformă automat sau chiar rotește o reprezentare bidimensională a unui obiect tridimensional. Forma este definită de o serie de puncte într-un spațiu absolut. Masivul iterativ aplică transformările liniare necesare pentru a calcula deflecția pe plăcile x și y ale unui tub cu raze catodice. O a treia coordonată de ieșire ar putea controla intensitatea spotului.

Soluția se bazează pe cele reprezentate în fig. 4.5. Fiecare coloană a masivului (cu excepția ultimei) este modelată de soluția din X4. Astfel se copiează către vecinul său prin canalul de transmisie orizontal fiecare valoare recepționată prin canalul de intrare orizontal.

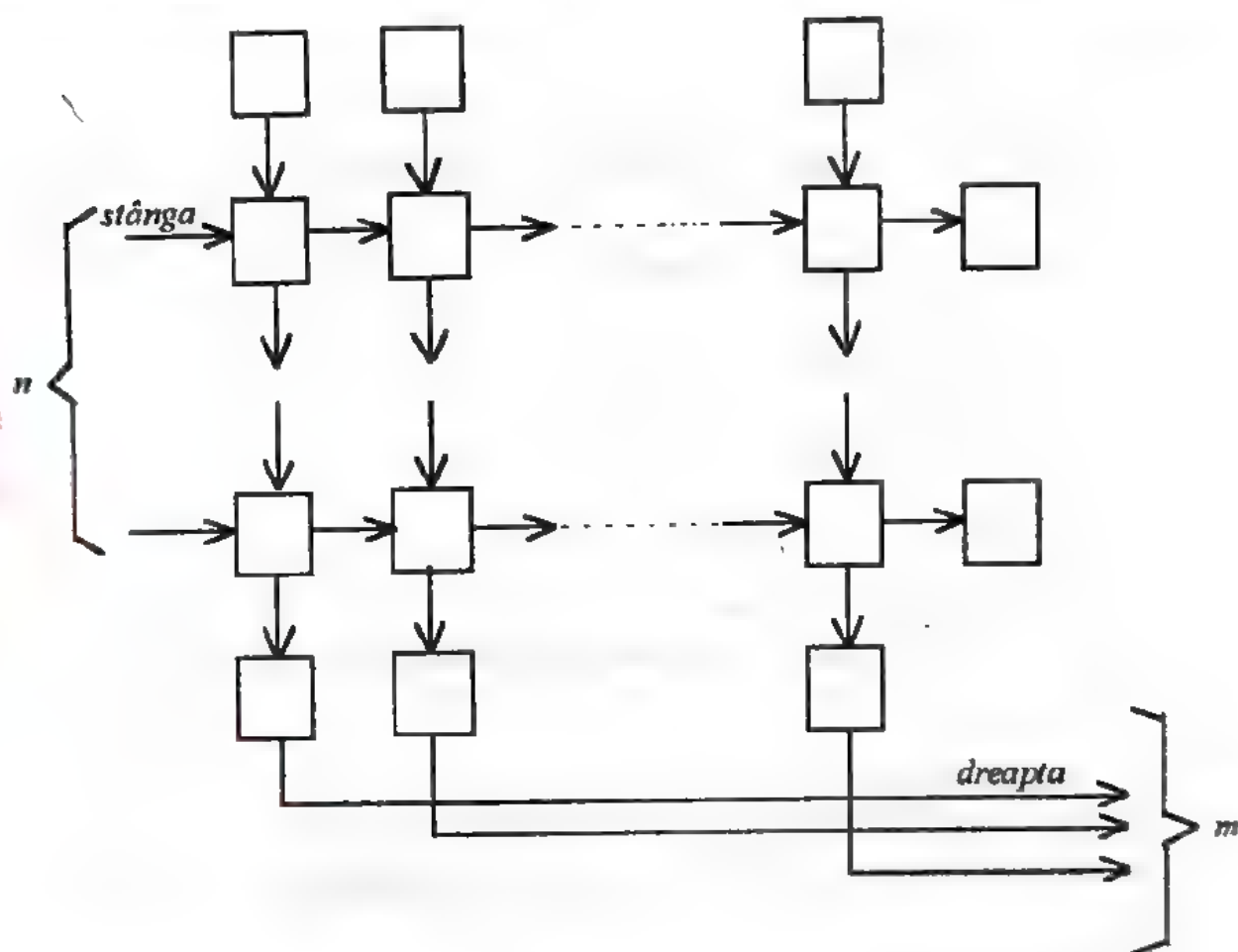


Figura 4.5

Procesele din marginea dreaptă pur și simplu renunță la valorile pe care le recepționează. Astfel, ar fi posibilă o economie de procesoare prin înglobarea funcțiilor celor marginale în acelea ale vecinilor.

Detaliile soluției sunt lăsate ca exercițiu. □

X6 Recepția pe canalul c poate fi interpretată ca succesiunea cifrelor unui număr natural C pornind de la cifra cea mai puțin semnificativă și exprimate în baza b . Definim valoarea numărului recepționat ca

$$C = \sum_{i \geq 0} c[i] \times b^i$$

unde $c[i] < b$ pentru toți i

Dat un multiplicator fix M (o cifră), transmisia pe canalul d este o succesiune de cifre ale produsului $M \times C$. Cifrele trebuie transmise cu o întârziere minimă. Să specificăm problema mai precis. Valoarea dorită a transmisiei d este

$$d = \sum_{i \geq 0} M \times c[i] \times b^i$$

Al j -lea element al lui d trebuie să fie cifra j care poate fi calculată cu formula

$$\begin{aligned} d[j] &= ((\sum_{i \geq 0} M \times c[i] \times b^i) \div b^j) \bmod b \\ &= (M \times C[j] + z_j) \bmod b \end{aligned}$$

unde $z_j = (\sum_{i < j} M \times c[i] \times b^i) \div b^j$

iar \div semnifică împărțirea întreagă.

z_j este transportul și se poate rapid dovedi că satisface definiția inductivă

$$\begin{aligned} z_0 &= 0 \\ z_{j+1} &= ((M \times C[j] + z_j) \div b) \end{aligned}$$

De aceea definim un proces $MULT1(z)$ care are ca parametru transportul z

$$MULT1(z) = c?x \rightarrow d!((M \times x + z) \bmod b) \rightarrow MULT1((M \times x + z) \div b)$$

Valoarea inițială pentru z este 0 astfel că soluția cerută este

$$MULT = MULT1(0)$$

□

X7 Problema este aceeași ca în X6 cu deosebirea că M este un număr cu mai multe cifre

$$M = \sum_{i < n} M_i \times b^i$$

Am văzut cum un singur procesor poate multiplica numere doar cu o singură cifră. De altfel și transmisia se produce cu o viteză care permite o multiplicare numai pe cifră. Prin urmare, cel puțin n procesoare sunt necesare. Vom face ca fiecare proces NOD_i să gestioneze o cifră M_i a multiplicatorului.

Baza soluției este tradiționalul algoritm manual pentru înmulțire cu mai multe cifre, cu deosebirea că sumele parțiale sunt adunate imediat la următorul rând al tabelui.



Figura 4.6

..... 153091	C	numărul ce se va recepționa
253	M	multiplicatorul
<hr/>		
.... 306182	$M_2 \times C$	calculat de NOD_2
.... 765455	$M_1 \times C$	calculat de NOD_1
..... 827275	$5 \times C$	
... 459273	$M_0 \times C$	calculat de NOD_0
... 732023	$M \times C$	

Nodurile sunt conectate așa cum se arată în fig. 4.6. Recepția originală are loc pe canalul c_0 și este propagată în stânga pe canalele c . Rezultatele parțiale sunt propagate spre dreapta prin canalele d iar rezultatul final este transmis pe d_0 . Din fericire, fiecare nod poate să comunice o cifră a rezultatului său înaintea comunicației cu vecinul din stânga. Mai mult, nodul cel mai din stânga poate fi definit să se comporte conform cu X6

$$NODE_{n-1}(z) = c_{n-1} ? x \rightarrow d_{n-1} ! (M_{n-1} \times x + z) \bmod b \\ \rightarrow NODE_{n-1}((M_{n-1} \times x + z) + b)$$

Nodurile rămase sunt la fel cu celelalte cu deosebirea că fiecare din ele pasează cifra recepționată vecinului său stâng și adună ce primește de la același vecin stâng la propriul transport. Pentru $k < n-1$

$$NODE_k(z) = c_k ? x \rightarrow d_k ! (M_k \times x + z) \bmod b \rightarrow c_{k+1} ! x \rightarrow d_{k+1} ? y \\ \rightarrow NODE_k(y + (M_k \times x + z) + b)$$

Întreaga rețea este definită

$$\|NODE_i(0) \quad i < n$$

□

X7 este un exemplu simplu dintr-o clasă de algoritmi ingenioși pentru rețele în care există un ciclu principal în graful orientat al canalelor de comunicație. Esența problemei a fost mult simplificată de presupunerea că multiplicatorul este cunoscut înainte și fixat pentru totdeauna. Într-o aplicație practică este mult mai potrivit ca acești parametri să trebuiască a fi recepționați pe același canal ca și datele propriu-zise și de asemenea să poată fi schimbați ori de câte ori e necesar. Desigur, această implementare necesită mult mai mare grijă dar și puțină ingeniozitate.

O metodă simplă de implementare este de a introduce un simbol special, de exemplu *reîncărcare* ca un indiciu că următorul număr sau numere trebuie tratate ca o schimbare de parametru, iar dacă numărul parametrilor este variabil trebuie de asemenea introdus simbolul *sfârșit_reîncărcare*.

X8 În acest exemplu avem aceeași situație ca în X4 cu excepția faptului că parametrii V_j trebuie reîncărcați cu numărul ce urmează imediat unui simbol *reîncărcare*. Definiția lui $MULT_{j+1}$ trebuie schimbată pentru a include multiplicatorul ca parametru

$$\begin{aligned} &MULT_{j+1} = stanga_j ? x \rightarrow \\ &\quad \text{if } x = \text{reîncărcare then } (stanga_j ? y \rightarrow MULT_{j+1}(y)) \\ &\quad \text{else } (mij_j ? y \rightarrow mij_{j+1}(v \times x + y) \rightarrow MULT_{j+1}(v)) \end{aligned}$$

□

4.4 Conducute

În acest paragraf ne vom concentra atenția la procesele cu numai două canale în alfabet, numite canal de recepție (intrare) *stânga* și canal de transmisie (ieșire) *dreapta*. Astfel de procese se numesc *conducute* și pot fi reprezentate ca în fig. 4.7.

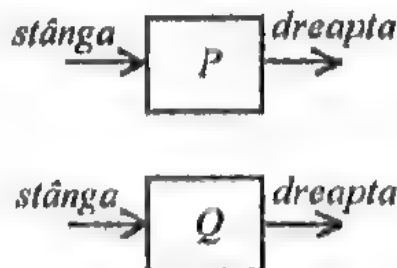


Figura 4.7

Procesele P și Q pot fi legate astfel încât canalul *dreapta* al lui P să fie conectat la canalul *stânga* al lui Q și secvența mesajelor transmise de P și recepționate de Q pe acest canal intern să fie mascat mediului lor. Rezultatul acestei conexiuni este notat

$$P \gg Q$$

și poate fi reprezentat ca în fig. 4.8.



Figura 4.8

Această diagramă de conexiune ne arată mascarea canalului de conexiune prin absența unui nume. De asemenea desenul arată că toate mesajele recepționate de canalul *stânga* al lui $(P \gg Q)$ sunt recepționate de P și toate mesajele emise pe canalul din *dreapta* a lui $(P \gg Q)$ sunt emise de Q . În final $(P \gg Q)$ este el însuși o conductă și poate fi legat în serie cu alte conducte

$$(P \gg Q) \gg R, (P \gg Q) \gg (R \gg S) \text{ etc.}$$

Din 4.4.1 L1 operatorul \gg este asociativ, astfel că pe viitor vom omite parantezele într-o astfel de succesiune.

Validitatea înlănțuirii proceselor prin \gg depinde evident de constrângerile de alfabet

$$\alpha(P \gg Q) = \alpha_{\text{stânga}}(P) \cup \alpha_{\text{dreapta}}(Q)$$

iar o altă constrângere afirmă despre canalele conectate că sunt capabile să transmită numai aceleași mesaje

$$\alpha_{\text{dreapta}}(P) = \alpha_{\text{stânga}}(Q)$$

Exemple

X1 O conductă care transmite fiecare valoare recepționată multiplicată cu patru (4.2 X2)

$$\text{4ORI} = \text{DUBLU} \gg \text{DUBLU}$$



X2 Un proces care recepționează cartele de 80 caractere și transmite conținutul lor împachetat în linii de câte 125 de caractere (4.2 X3, X4)

DESPACHET ➤ *ÎMPACHET*

Acest proces este cam dificil de scris utilizând tehnicile de programare structurată convenționale deoarece nu este clar dacă bucla principală este bine să se facă pentru recepția fiecărei cartele sau pentru transmisia fiecărei linii. Problema este cunoscută de Michael Jackson ca *dezacordul de structură*. Soluția dată mai sus conține o buclă separată în fiecare din cele două procese, care se potrivește foarte bine structurii problemei originale. □

X3 Avem aceeași situație ca în X2 cu deosebirea că fiecare pereche de asteriscuri consecutive este înlocuită cu "↑" (4.2 X5)

DESPACHET ➤ *ELIMIN* ➤ *ÎMPACHET*

Într-un program secvențial convențional această schimbare minoră în specificație ar putea cauza probleme severe. Este ușor de evitat astfel de probleme prin simpla inserare a unui nou proces. Acest fel de modularitate a fost introdus și exploatat de proiectanții de sisteme de operare. □

X4 Aceeași situație ca în X2 cu excepția faptului că cititul cartelelor poate continua când imprimanta este neoperațională, iar imprimatul poate continua când cititorul este neoperațional (4.2 X9)

DESPACHET ➤ *BUFFER* ➤ *ÎMPACHET*

Bufferul ține caracterele care au fost produse de procesul *DESPACHET* dar nu au fost încă consumate de procesul *ÎMPACHET*. Ele vor fi disponibile pentru recepția lui *ÎMPACHET*, timp în care procesul *DESPACHET* este temporar întârziat. Bufferul netezește astfel variațiile temporare în viteza de producție și consum. Totuși soluția nu poate rezolva problema nepotrivirii pe termen lung a vitezelor de producție și consum. Dacă cititorul de cartele este în medie mai lent decât imprimanta, bufferul va fi aproape tot timpul gol și nu se va realiza nici un efect de potrivire. Dacă cititorul este mai rapid, bufferul se va extinde la infinit până când consumă tot spațiul disponibil de stocare. □

X5 Pentru a evita expansiunea nedorită a bufferelor, se obișnuiește să se limiteze numărul mesajelor stocate. Chiar și un singur buffer realizat de procesul *COPIE* (4.2 X1) poate fi adecvat. Iată o versiune a lui X4 care citește în avans o cartelă la intrare și stochează o linie de imprimat la ieșire

$COPIE \gg DESPACHET \gg ÎMPACHET \gg COPIE$

De notat că alfabetele celor 2 procese $COPIE$ sunt diferite, lucru care trebuie înțeles din contextul în care sunt plasate. \square

X6 Un buffer dublu care acceptă până la două mesaje înainte de a se face transmisia primului

$COPIE \gg COPIE$

\square

Comportarea sa este aproape similară cu cea a lui $LANT2$ (2.6 X4) și chiar $ATS2$ (1.1.3 X6).

4.4.1 Legi

Cea mai utilă proprietate algebrică a înlănțuirii este asociativitatea

$$L1 \quad P \gg (Q \gg R) = (P \gg Q) \gg R$$

Restul legilor prezintă cum recepția și transmisia pot fi implementate printr-o conductă. Ele permit ca descrierile proceselor să fie simplificate la o formă de execuție simbolică. De exemplu, dacă procesul din stânga lui \gg începe cu transmisia unui mesaj v prin canalul *dreapta* și procesul din dreapta lui \gg începe cu recepția de la *stânga*, mesajul v este transmis de la primul proces la cel de-al doilea. Totuși comunicația ce are loc este mascată, așa cum se arată în legea

$$L2 \quad (dreapta!v \rightarrow P) \gg (stânga?y \rightarrow Q(y)) = P \gg Q(v)$$

Dacă unul din procese este determinat să comunice cu celălalt, dar celălalt este pregătit să comunice extern, are loc mai întâi comunicația externă iar comunicația internă este amânată până la o ocazie ulterioară

$$L3 \quad (dreapta!v \rightarrow P) \gg (dreapta!w \rightarrow Q) = dreapta!w \rightarrow ((dreapta!v \rightarrow P) \gg Q)$$

$$L4 \quad (stânga?x \rightarrow P(x)) \gg (stânga?y \rightarrow Q(y)) \\ = stânga?x \rightarrow (P(x) \gg (stânga?y \rightarrow Q(y)))$$

Dacă ambele procese sunt pregătite pentru comunicație externă, oricare din ele poate avea loc mai întâi

$$L5 \quad (stânga?x \rightarrow P(x)) \gg (dreapta!w \rightarrow Q) = (stânga?x \rightarrow (P(x) \gg (dreapta!w \rightarrow Q))) \\ | dreapta!w \rightarrow ((stânga?x \rightarrow P(x)) \gg Q)$$

Legea L5 este de asemenea valabilă când operatorul \gg este înlocuit cu $\gg R$, deoarece conductele din interiorul unui lanț nu pot comunica direct cu mediul

$$\begin{aligned} \text{L6 } (st\grave{a}nga?x \rightarrow P(x)) \gg R \gg (dreapta!w \rightarrow Q) = \\ (st\grave{a}nga?x \rightarrow (P(x) \gg R \gg (dreapta!w \rightarrow Q))) | dreapta!w \rightarrow ((st\grave{a}nga?x \rightarrow P(x)) \gg R \gg Q) \end{aligned}$$

Generalizări analoage pot fi făcute altor legi.

L7 Dacă R este un lanț de procese, toate pornind cu transmisia prin dreapta

$$R \gg (dreapta!w \rightarrow Q) = dreapta!w \rightarrow (R \gg Q)$$

L8 Dacă R este un lanț de procese, toate pornind cu recepție din stânga

$$(st\grave{a}nga?x \rightarrow P(x)) \gg R = st\grave{a}nga?x \rightarrow (P(x) \gg R)$$

Exemple

X1 Să definim

$$R(y) = (dreapta!y \rightarrow COPIE) \gg COPIE$$

$$\begin{aligned} \text{Astfel } R(y) &= (dreapta!y \rightarrow COPIE) \gg (st\grave{a}nga?x \rightarrow dreapta!x \rightarrow COPIE) && \text{definiția lui COPIE} \\ &= COPIE \gg (dreapta!y \rightarrow COPIE) && \text{L2} \quad \square \end{aligned}$$

X2 $COPIE \gg COPIE$

$$\begin{aligned} &= (st\grave{a}nga?x \rightarrow dreapta!x \rightarrow COPIE) \gg COPIE && \text{definiția lui COPIE} \\ &= (st\grave{a}nga?x \rightarrow ((dreapta!x \rightarrow COPIE) \gg COPIE)) && \text{L4} \\ &= st\grave{a}nga?x \rightarrow R(x) && \text{definiția lui R(x)} \quad \square \end{aligned}$$

X3 Din ultima linie a lui X1 deducem

$$\begin{aligned} R(y) &= (st\grave{a}nga?x \rightarrow dreapta!x \rightarrow COPIE) \gg (dreapta!y \rightarrow COPIE) \\ &= (st\grave{a}nga?x \rightarrow ((dreapta!x \rightarrow COPIE) \gg (dreapta!y \rightarrow COPIE))) \\ &\quad | dreapta!y \rightarrow (COPIE \gg COPIE) && \text{L5} \\ &= (st\grave{a}nga?x \rightarrow dreapta!y \rightarrow R(x)) && \text{L3} \\ &\quad | dreapta!y \rightarrow st\grave{a}nga?x \rightarrow R(x) && \text{X2} \end{aligned}$$

Aceasta ne arată că un buffer dublu, după primirea primului mesaj este pregătit atât să transmită acel mesaj cât și să recepționeze un al doilea înainte

de a face transmisia. Raționamentele pentru cele de mai sus sunt foarte asemănătoare cu cele din 2.3.1 X1. \square

4.4.2 Implementare

În implementarea lui $P \gg Q$ se disting trei cazuri

- (1) Dacă comunicația poate avea loc pe canalul de conexiune intern, aceasta se produce imediat, fără considerarea mediului extern. Dacă este posibil o secvență infinită de astfel de comunicații, procesul diverge (paragraful 3.5.2.).
- (2) Altfel, dacă mediul este interesat în comunicația pe canalul stâng, aceasta este tratată de P .
- (3) Dacă mediul este interesat de comunicația prin canalul drept, aceasta este tratată de Q .

Pentru explicitarea operațiilor de recepție și transmisie, vezi paragraful 4.2.1

```
lanț(P,Q)=
if P("dreapta")≠"BLIP" ∧ Q("stânga")≠"BLIP"
then lanț(cdr(P("dreapta")), Q("stânga")(car(P("dreapta"))))    caz (1)
else λx. if x="dreapta
then if Q("dreapta")="BLIP" then "BLIP
      else cons(car(Q("dreapta")),
                lanț(P, cdr(Q("dreapta"))))    caz (3)
else if x="stânga
then if P(x)="BLIP" then "BLIP
      else λy.lanț(P("stânga") (y),Q)    caz (2)
else "BLIP
```

4.4.3 Blocaj prin ciclare infinită

Operatorul de înlănțuire conectează două procese printr-un singur canal, de aceea nu există riscul de blocaj. Dacă atât P cât și Q sunt bucle, atunci nici $(P \gg Q)$ nu se va opri. Din nefericire, există un nou pericol și anume acela ca procesele P și Q să-și petreacă tot timpul comunicând între ele, astfel încât $(P \gg Q)$ să nu mai comunice cu lumea externă. Acest caz de divergență (paragraful 3.5.1, 3.8) este ilustrat de exemplul banal

```
P=(dreapta!1→P)
Q=(stânga!x→Q)
```

$(P \gg Q)$ este evident un proces inutil. El este chiar mai rău decât *STOP* prin faptul că o buclă fără sfârșit poate consuma resurse de calcul fără limită și fără a produce ceva. Un exemplu mai puțin banal este $(P \gg Q)$ unde

$$P = (\text{dreapta}! 1 \rightarrow P \mid \text{stânga}? x \rightarrow P1(x))$$

$$Q = (\text{stânga}? x \rightarrow Q \mid \text{dreapta}! 1 \rightarrow Q1)$$

În acest exemplu, divergența provine din simpla posibilitate de comunicare internă infinită. Divergența există chiar dacă este oferită alternativa unei comunicări externe prin canalele *stânga* și *dreapta* cu orice ocazie posibilă și chiar dacă după o astfel de comunicare externă, următoarea comportare a lui $(P \gg Q)$ n-ar diverge.

O metodă simplă de a demonstra că $(P \gg Q)$ este lipsit de blocajul prin ciclare infinită este de a arăta că P este cu *gardă-stângă* în sensul că nu poate transmite niciodată o serie infinită de mesaje prin *dreapta* fără a se interpune recepții prin *stânga*. Pentru a ne asigura de aceasta, trebuie să dovedim că lungimea secvenței transmise prin *dreapta* este oricând mărginită superior de o funcție bine definită f a secvenței de valori recepționate prin *stânga*, sau mai formal definim

$$P \text{ este cu gardă-stângă} \equiv \exists f. P \text{ sat } (\# \text{dreapta} \leq f(\text{stânga}))$$

Proprietatea de gardă-stângă este adesea evidentă din scrierea lui P .

L1 Dacă orice recursivitate utilizată în definirea lui P este cu gardă datorită recepției prin *stânga*, atunci P este cu gardă-stângă.

L2 Dacă P este cu gardă-stângă, atunci $(P \gg Q)$ este liber de blocaj prin ciclare infinită.

Exact același raționament se aplică pentru a pune în evidență garda-dreaptă a celui de-al doilea operand al lui \gg .

L3 Dacă Q este cu gardă-dreaptă, atunci $(P \gg Q)$ este liberă de blocaj prin ciclare infinită.

Exemple

X1 Următoarele procese sunt cu gardă-stângă datorită lui L1 (4.1 X1, X2, X5, X9)

COPIE, DUBLU, ELIMIN, BUFFER

□

X2 Următoarele procese sunt cu gardă-stângă potrivit definiției originale, din cauză că

DESPACHET sat $\#dreapta \leq \#(\backslash stânga)$

ÎMPACHET sat $\#dreapta \leq \#stânga$ □

X3 *BUFFER* nu este cu gardă-dreaptă deoarece poate recepționa arbitrar de multe mesaje în stânga fără să le transmită vreodată în dreapta. □

4.4.4 Specificații

Specificarea unei conduite poate fi exprimată adesea ca o expresie $S(stânga, dreapta)$ corespunzătoare secvenței de mesaje recepționate pe canalul din stânga și secvenței de mesaje transmise prin dreapta. Când două conduite sunt conectate în serie, secvența *dreapta* produsă de operandul stâng este egalată de secvența *stânga* consumată de operandul drept. Această secvență comună este apoi mascată. Tot ceea ce se știe despre secvența mascată este că ea există. Dar de asemenea vrem să prevenim riscul blocării prin ciclare infinită. De aceea prezentăm regula

L1 Dacă P sat $S(stânga, dreapta)$
 și Q sat $T(stânga, dreapta)$
 și dacă P este cu gardă-stângă sau Q cu gardă-dreaptă
 atunci $(P \gg Q)$ sat $\exists s. S(stânga, s) \wedge T(s, dreapta)$

Această regula afirmă că relația între *stânga* și *dreapta* promovată de $(P \gg Q)$ este o compunere relațională normală între relația lui P cu relația lui Q . Deoarece operatorul \gg nu poate introduce blocaj în conduite ne putem permite să ometem raționamentul pentru refuzuri.

Exemple

X1 *DUBLU* sat $\overset{1}{dreapta} \leq \overset{1}{dublu^*}(stânga)$
DUBLU este cu gardă-stângă și cu gardă-dreaptă

Astfel $(DUBLU \gg DUBLU)$ sat $\exists s. (\overset{1}{s} \leq \overset{1}{dublu^*}(stânga) \wedge \overset{1}{dreapta} \leq \overset{1}{dublu^*}(s))$
 $\overset{2}{=} dreapta \leq \overset{2}{dublu^*}(dublu^*(stânga))$
 $\overset{2}{=} dreapta \leq \overset{2}{4ori^*}(stânga)$ □

X2 Să aplicăm recursivitatea împreună cu \gg pentru a da o altă definiție unui buffer

$$BUFF = \mu X. (st\acute{a}nga?x \rightarrow (X \triangleright (dreapta!x \rightarrow COPIE)))$$

Dorim să dovedim că

$$BUFF \text{ sat } (dreapta \leq st\acute{a}nga)$$

Presupunem că

$$X \text{ sat } \# st\acute{a}nga \geq n \vee dreapta \leq st\acute{a}nga$$

Știm că

$$COPIE \text{ sat } dreapta \leq st\acute{a}nga$$

Rezultă că $(dreapta!x \rightarrow COPIE) \text{ sat } ((dreapta = st\acute{a}nga = \Diamond \vee (dreapta \geq \langle x \rangle \wedge dreapta' \leq st\acute{a}nga)) \Rightarrow dreapta \leq \langle x \rangle \wedge st\acute{a}nga)$

Deoarece operandul drept este cu gardă-dreaptă, din L1 și presupunând că

$$\begin{aligned} (X \triangleright (dreapta!x \rightarrow COPIE)) \text{ sat } (\exists s. (\#st\acute{a}nga \geq n \vee s \leq st\acute{a}nga) \\ \wedge dreapta \leq \langle x \rangle \wedge s) \\ \Rightarrow (\#st\acute{a}nga \geq n \vee dreapta \leq \langle x \rangle \wedge st\acute{a}nga) \end{aligned}$$

deducem $st\acute{a}nga?x \rightarrow (...) \text{ sat } dreapta \rightarrow st\acute{a}nga = \Diamond$
 $\vee (st\acute{a}nga > \Diamond \wedge (\#st\acute{a}nga \geq n \vee dreapta \leq \langle st\acute{a}nga_0 \rangle \wedge st\acute{a}nga'))$
 $\Rightarrow \#st\acute{a}nga \geq n+1 \vee dreapta \leq st\acute{a}nga$

Concluzia dorită rezultă din legea pentru procese recursive (3.7.1 L8). Legea mai simplă (1.10.2 L6) nu poate fi folosită din cauză că recursivitățile nu sunt cu gardă evidentă. \square

4.4.5 Buffere și protocoale

Un buffer este un proces care transmite în partea dreaptă exact aceeași secvență de mesaje pe care a primit-o în stânga, posibil cu o întârziere. Astfel că, dacă nu este gol este totdeauna gata să transmită în dreapta. Mai formal, definim un buffer ca un proces care nu se oprește niciodată, care este liber de blocaj prin ciclare infinită și care satisface specificația

$$\begin{aligned} P \text{ sat } (dreapta \leq st\acute{a}nga) \\ \wedge (\text{if } dreapta = st\acute{a}nga \text{ then } st\acute{a}nga \notin ref \text{ else } dreapta \notin ref) \end{aligned}$$

Aici *ceref* înseamnă că procesul nu poate refuza comunicarea pe canalul *c* (paragraful 3.7., 3.4). Urmează că toate bufferele sunt cu gardă-stângă.

Exemplu

X1 Următoarele procese sunt buffere

COPIE, (*COPIE* \triangleright *COPIE*), *BUFF*, *BUFFER*

□

Bufferele sunt evident utile în stocarea informației care urmează a fi procesată. Dar sunt chiar și mai utile ca specificații pentru comportarea dorită a unui protocol de comunicație care se intenționează să livreze mesaje în aceeași ordine în care au fost primite. Un astfel de protocol constă din două procese, un transmițător *T* și un receptor *R* care sunt conectate în serie (*T* \triangleright *R*). Dacă un protocol este corect este clar că (*T* \triangleright *R*) trebuie să fie un buffer.

În practică, suportul fizic care conectează transmițătorul de receptor este suficient de lung și mesajele care sunt trimise prin el pot fi deteriorate sau pierdute. Astfel, comportarea cablului însuși poate fi modelată de un proces *FIRE*, care se comportă nu tocmai ca un buffer. Este sarcina proiectantului de protocol să se asigure că în ciuda proastei comportări a suportului (cablu), sistemul ca întreg se comportă ca un buffer, adică

(*T* \triangleright *FIRE* \triangleright *R*) este un buffer

Un protocol este de obicei realizat dintr-un număr de straturi (*T*₁, *R*₁), (*T*₂, *R*₂), ..., (*T*_{*n*}, *R*_{*n*}), fiecare utilizând stratul anterior drept mediul său de comunicare.

$$T_n \triangleright \dots \triangleright (T_2 \triangleright (T_1 \triangleright FIRE \triangleright R_1) \triangleright R_2) \triangleright \dots \triangleright R_n$$

Desigur, când protocolul este implementat în practică, toți transmițătorii sunt strânși într-unul singur la un capăt și toți receptorii la celălalt, în concordanță cu modificarea asocierii în paranteze

$$(T_n \triangleright \dots \triangleright T_2 \triangleright T_1) \triangleright FIRE \triangleright (R_1 \triangleright R_2 \triangleright \dots \triangleright R_n)$$

Asociativitatea pentru \triangleright garantează că această regrupare nu schimbă comportarea sistemului.

În practică, protocoalele trebuie să fie mult mai complicate ca acesta. Unidirecționalitatea transferului mesajelor nu este adecvată pentru realizarea unei comunicații fiabile pe un cablu nefiabil. De aceea este necesar să adăugăm canale pentru comunicația inversă permițând receptorului să trimită

înapoi semnale de confirmare a bunei recepționări a mesajelor astfel ca mesajele neconfirmate să poată fi retransmise.

Următoarele legi se dovedesc utile în dovedirea corectitudinii protocoalelor. Ele se datoresc lui A. W. Roscoe.

L1 Dacă P și Q sunt buffere,

sunt și $(P \gg Q)$ și $(st\grave{a}nga?x \rightarrow (P \gg (dreapta!x \rightarrow Q)))$

L2 Dacă $(T \gg R) = (st\grave{a}nga?x \rightarrow (T \gg (dreapta!x \rightarrow R)))$ atunci $(T \gg R)$ este un buffer.

Următoarea lege este o generalizare a lui L2

L3 Dacă pentru o funcție f și toți z

$(T(z) \gg R(z)) = (st\grave{a}nga?x \rightarrow (T(f(x,z)) \gg (dreapta!x \rightarrow R(f(x,z)))))$

atunci $T(z) \gg R(z)$ este un buffer pentru toți z .

Exemple

X1 Următoarele procese sunt buffere datorită lui L1

$COPIE \gg COPIE, BUFFER \gg COPIE, COPIE \gg BUFFER,$
 $BUFFER \gg BUFFER$

□

X2 S-a arătat în 4.4.1 X1 și X2 că

$(COPIE \gg COPIE) = (st\grave{a}nga?x \rightarrow (COPIE \gg (dreapta!x \rightarrow COPIE)))$

Din L2 rezultă că membrul stâng este un buffer.

□

X3 Codarea în fază.

Un codor în fază este un proces T care recepționează un flux de biți și transmite $\langle 0, 1 \rangle$ pentru fiecare 0 recepționat și $\langle 1, 0 \rangle$ pentru fiecare 1 recepționat. Un decodor R inversează această translație

$T = st\grave{a}nga?x \rightarrow dreapta!x \rightarrow dreapta!(1-x) \rightarrow T$

$R = st\grave{a}nga?x \rightarrow st\grave{a}nga!y \rightarrow \text{if } y=x \text{ then } EROARE \text{ else } (dreapta!x \rightarrow R)$

unde procesul $EROARE$ este lăsat nedefinit.

Dorim să dovedim că $T \gg R$ este un buffer, prin L2

$$\begin{aligned}
(T \triangleright R) &= st\acute{a}nga?x \rightarrow ((dreapta!x \rightarrow dreapta!(1-x) \rightarrow T)) \\
&\triangleright (st\acute{a}nga?x \rightarrow st\acute{a}nga?y \rightarrow \text{if } y=x \text{ then } EROARE \text{ else } (dreapta!x \rightarrow R))) \\
&= st\acute{a}nga?x \rightarrow (T \triangleright \text{if } (1-x)=x \text{ then } EROARE \text{ else } (dreapta!x \rightarrow R)) \\
&= st\acute{a}nga?x \rightarrow (T \triangleright (dreapta!x \rightarrow R))
\end{aligned}$$

De aceea $(T \triangleright R)$ este un buffer prin L2. □

X4 Schimbarea unui bit.

Transmițătorul T reproduce cu fidelitate biții recepționați prin canalul din stânga spre canalul din dreapta cu excepția cazului când după trei de 1 consecutivi care au fost transmiși, schimbă un bit în 0. Astfel intrarea 01011110 este transmisă ca 010111010. Receptorul R trebuie să facă operația inversă. Astfel, trebuie arătat că $(T \triangleright R)$ este un buffer. Construcția lui T și a lui R și dovedirea corectitudinii sunt lăsate ca exerciții. □

X5 Partajarea liniei.

Se dorește copierea unor date de la un canal *stânga1* la *dreapta1* și de la *stânga2* la *dreapta2*. Aceasta poate fi cel mai ușor realizat prin două protocoale disjuncte fiecare utilizând alt cablu. Din nefericire, numai un singur cablu *mij* este disponibil și el trebuie folosit de ambele fluxuri de date ca în fig. 4.9.



Figura 4.9

Mesajele recepționate de T trebuie marcate înaintea transmiterii prin *mij* iar R trebuie să le demarcheze și să le transmită pe canalul corespunzător din dreapta.

$$\begin{aligned}
T &= (st\acute{a}nga1?x \rightarrow mij!marc1(x) \rightarrow T) \mid (st\acute{a}nga2?y \rightarrow mij!marc2(y) \rightarrow T) \\
R &= (mij?z \rightarrow \text{if } marc(z)=1 \text{ then } (dreapta1!demarc(z) \rightarrow R) \\
&\quad \text{else } (dreapta2!demarc(z) \rightarrow R))
\end{aligned}$$

Soluția este destul de nesatisfăcătoare. Dacă două mesaje sunt recepționate pe *stânga1*, dar receptorul nu este încă gata pentru ele, întregul sistem va trebui să aștepte iar transmisia între *stânga2* și *dreapta2* este serios întârziată. Dacă introducem buffere pe canale, amânăm numai puțin problema. Soluția corectă este de a introduce un alt canal în direcția inversă iar R să trimită semnale înapoi lui T oprind transmiterea mesajelor pe direcția pentru care pare să fie

cerere puțină. Această problemă este cunoscută drept *controlul fluxului* (*flow control*). \square

4.5 Subordonarea

Fie P și Q procese cu

$$\alpha P \subseteq \alpha Q$$

În combinația $(P||Q)$ fiecare acțiune a lui P poate apare numai când Q îi permite să apară. Cu toate acestea Q se poate angaja independent în acțiunile lui $(\alpha Q - \alpha P)$ fără cunoștința și permisiunea partenerului său P . Astfel P este pentru Q un fel de proces slave sau subordonat, în timp ce Q se comportă ca un proces principal sau master. Când comunicațiile între un proces subordonat și unul principal trebuie mascate mediului lor comun folosim notația asimetrică

$$P//Q$$

Utilizând operatorul de mascare, aceasta poate fi definită

$$P//Q = (P||Q) \setminus \alpha P$$

Această notație se folosește numai când $\alpha P \subseteq \alpha Q$ și atunci

$$\alpha(P//Q) = (\alpha Q - \alpha P)$$

Se obișnuiește să i se dea procesului subordonat un nume, fie m , care este folosit în procesul principal pentru toate interacțiunile cu subordonatul. Tehnica de numire a proceselor descrisă în paragraful 2.6.2 poate fi rapid extinsă la procese comunicante prin introducerea numelor de canal compuse. Acestea iau forma $m.c$, unde m este un nume de proces și c este unul din canalele sale. Fiecare comunicație pe acest canal este un triplet

$$m.c.v$$

unde $\alpha m.c(m:P) = \alpha c(P)$ și $v \in \alpha c(P)$

În construcția $(m:P//Q)$, Q comunică cu P de-a lungul canalelor cu nume compuse de forma $m.c$ și $m.d$. Avem în vedere de asemenea că P folosește canalele corespunzătoare simple c și d pentru aceleași comunicații. Astfel, de exemplu

$$(m:(c!v \rightarrow P) \backslash (m.c?x \rightarrow Q(x))) = (m:P/Q(v))$$

Deoarece toate aceste comunicații sunt mascate mediului numele m nu poate fi aflat din exterior servind drept etichetă *locală* pentru procesul subordonat.

Subordonarea poate fi imbricată, de exemplu

$$(n:(m:P/Q) \backslash R)$$

În acest caz toate aparițiile evenimentelor implicând numele m sunt mascate înainte ca numele n să fie atașat evenimentelor rămase, ce se află în alfabetul lui Q dar nu și în al lui P . Nu există nici o modalitate prin care R să poată comunica direct cu P sau să știe de existența lui P sau a numelui său m .

Exemple

X1 $dub:DUBLU \backslash Q$

(pentru $DUBLU$ vezi 4.2 X2)

Procesul subordonat se comportă ca o simplă subrutină apelată din procesul principal Q . În interiorul lui Q , valoarea $2 \times e$ poate fi obținută printr-o transmisie succesivă a argumentului e pe canalul din stânga lui dub și recepția rezultatului pe canalul din dreapta

$$dub.st\acute{a}nga!e \rightarrow (dub.dreapta?x \rightarrow \dots)$$

□

X2 O subrutină poate utiliza o alta ca subordonată și acest lucru îl face de câteva ori

$$4ORI = (dub:DUBLU \backslash (\mu X.st\acute{a}nga?x \rightarrow dub.st\acute{a}nga!x \rightarrow \\ dub.dreapta?y \rightarrow dub.st\acute{a}nga!y \rightarrow dub.dreapta?z \rightarrow dreapta!z \rightarrow X))$$

Această rutină însăși este proiectată să fie utilizată ca o subrutină

$$4ori:4ORI \backslash Q$$

Această versiune a lui $4ORI$ este similară cu cea din 4.4 X1 dar nu are același efect de dublă bufferare. □

X3 O variabilă de program convenționabilă numită m poate fi modelată ca un proces subordonat

$$m:VAR \backslash Q$$

În interiorul procesului principal Q , valoarea lui m poate fi asignată, citită și actualizată prin recepție și transmisie, așa cum se descrie în 2.6.2 X2.

$m:=3;P$ este implementat prin $(m.st\acute{a}nga!3 \rightarrow P)$
 $x:=m;P$ este implementat prin $(m.dreapta?x \rightarrow P)$
 $m:=m+3;P$ este implementat prin $(m.dreapta?y \rightarrow m.st\acute{a}nga!(y+3) \rightarrow P)$ \square

Un proces subordonat poate fi folosit la implementarea unei structuri de date cu o comportare mai elaborată decât cea a unei simple variabile.

X4 ($q:BUFFER/Q$) (vezi 4.2 X9)

Procesul subordonat are rolul unei cozi nemărginite numită q . În interiorul lui Q , transmisia lui $q.st\acute{a}nga!v$ adaugă v la un capăt al cozii și $q.dreapta?y$ elimină un element din celălalt capăt și-i dă valoarea sa lui y . Dacă coada este goală, ea nu va răspunde și sistemul se va bloca. \square

X5 O stivă cu numele st este declarată

$st:STIV\acute{A}/Q$ (vezi 4.2 X10)

În interiorul procesului principal Q , $st.st\acute{a}nga!v$ poate fi utilizată pentru a depune valoarea v în stivă și $st.dreapta?x$ va extrage valoarea din vârf. Pentru a putea gestiona situația când stiva este goală, poate fi utilizată o construcție alternativă

$(st.dreapta?x \rightarrow Q1(x) | st.gol \rightarrow Q2)$

Dacă stiva nu este goală, este selectată prima alternativă. Dacă este goală, blocajul este evitat și este selectată a doua alternativă. \square

Un proces subordonat cu mai multe canale poate fi utilizat de mai multe procese concurente în condițiile când nu utilizează același canal.

X6 Un proces Q intenționează să comunice un flux de valori lui R . Aceste valori trebuie bufferate de un proces buffer subordonat numit b astfel încât transmiterea de la Q să nu fie întârziată când R nu este gata pentru recepție. Q utilizează canalul $b.st\acute{a}nga$ pentru transmisia sa și R utilizează $b.dreapta$ pentru recepția sa

$(b:BUFFER(Q||R))$

De remarcat că dacă R încearcă să recepționeze de la un buffer gol, sistemul nu se va bloca în mod necesar. R pur și simplu va fi întârziat până când Q va transmite următoarea valoare bufferului. (Dacă Q și R comunică cu bufferul pe același canal, atunci acel canal trebuie să fie în alfabetul amândorura și definiția lui \parallel ar cere ca ele să comunice totdeauna simultan aceeași valoare – ceea ce ar fi complet greșit). \square

Operatorul de subordonare poate fi folosit pentru a defini subrutine prin recursivitate. Fiecare nivel de recursivitate (cu excepția ultimului) declară o subrutină locală *nouă* care gestionează apelurile recursive în continuare.

X7 Factorial

$$FAC = \mu X. st\acute{a}nga?n \rightarrow (if\ n=0\ then\ (dreapta!1 \rightarrow X) \\ else\ (f.X / (f.st\acute{a}nga!(n-1) \rightarrow f.dreapta?y \rightarrow dreapta!(n \times y) \rightarrow X)))$$

Subrutina FAC utilizează canalele $st\acute{a}nga$ și $dreapta$ pentru a comunica parametri și rezultatele proceselor sale apelante. Ea folosește canalele $f.st\acute{a}nga$ și $f.dreapta$ pentru a comunica cu procesul subordonat numit f . Din această perspectivă ea este similară subrutinei $4ORI$ (X2). Singura diferență este că procesul subordonat scris este izomorf lui FAC însuși. \square

Exemplul de mai sus este un caz familiar și plictisitor de recursivitate exprimată într-un cadru notațional nefamiliar și cam complex. O idee mai puțin obișnuită este de a utiliza recursivitatea împreună cu subordonarea pentru a implementa o structură nemărginită de date. Fiecare nivel al recursivității reține o singură componentă a structurii și declară o *nouă* structură de date locală subordonată pentru a trata restul.

X8 Mulțime finită nemărginită

Un proces care implementează o mulțime recepționează membrii săi pe canalul său din stânga. După fiecare recepție, transmite un DA dacă a mai fost recepționată deja această valoare și NU altfel. Construcția este foarte similară cu mulțimea din 2.6.2 X4 cu excepția faptului că va reține mesaje de orice tip

$$MULTIME = st\acute{a}nga?x \rightarrow dreapta!NU \rightarrow (rest.MULTIME/BUCL\acute{A}(x))$$

$$unde\ BUCL\acute{A}(x) = \mu X. st\acute{a}nga?y \rightarrow (if\ y=x\ then\ (dreapta!DA \rightarrow X) \\ else\ (rest.st\acute{a}nga!y \rightarrow rest.dreapta?z \rightarrow dreapta!z \rightarrow X))$$

Mulțimea inițială este vidă. De aceea la recepția primului element x se va transmite NU . Apoi se declară un proces subordonat numit $rest$ care va reține toate elementele mulțimii cu excepția lui x . $BUCL\acute{A}$ este proiectat să re-

cepționeze elementele ulterioare ale mulțimii. Dacă noul element recepționat este egal cu x , răspunsul *DA* este trimis înapoi imediat pe canalul din dreapta. Altfel, noul element este acceptat pentru a fi reținut de *rest*. Se observă că oricare ar fi răspunsul (*DA* sau *NU*) trimis înapoi de *rest* se trece mai departe și *BUCLĂ* se repetă. \square

X9 Arbore binar

O reprezentare mai eficientă a unei mulțimi este aceea sub formă de arbore binar care se bazează pe o ordine totală \leq dată între elementele sale. Fiecare nod reține un element nou inserat și declară doi arbori subordonați. Unul va reține elementele mai mici decât rădăcina și celălalt elementele mai mari. Specificarea externă a arborelui este aceeași ca și în X8

$$\begin{aligned} \text{ARBORE} = & \text{st\u0103nga?}x \rightarrow \text{dreapta!}NU \rightarrow \\ & (\text{mai_mic:ARBORE}) \parallel (\text{mai_mare:ARBORE} \parallel \text{BUCL\u0103}) \end{aligned}$$

Proiectarea lui *BUCLĂ* este lăsată ca exercițiu. \square

4.5.1 Legi

Următoarele legi evidente guvernează comunicațiile între un proces și subordonații lui. Prima lege descrie mascarea comunicației în fiecare, sensul între procesul principal și subordonat

$$\text{L1A } (m:(c?x \rightarrow P(x))) \parallel (m.c!v \rightarrow Q) = (m:P(v)) \parallel Q$$

$$\text{L1B } (m:(d!v \rightarrow P)) \parallel (m.d?x \rightarrow Q(x)) = (m:P) \parallel Q(v)$$

Dacă b este un canal neetichetat cu m , procesul principal poate comunica prin b fără afectarea subordonatului

$$\text{L2 } (m:P \parallel (b!e \rightarrow Q)) = (b!e \rightarrow (m:P \parallel Q))$$

Singurul proces capabil de a face alegerea pentru procesul subordonat este procesul principal

$$\text{L3 } (m:(c?x \rightarrow P1(x) \mid d?y \rightarrow P2(y))) \parallel (m.c!v \rightarrow Q) = (m:P1(v) \parallel Q)$$

Dacă două procese subordonate au același nume, unul dintre ele este inaccesibil

$$\text{L4 } m:P \parallel (m:Q/R) = (m:Q/R)$$

De obicei, ordinea în care procesele subordonate sunt scrise nu contează

L5 Dacă n și m sunt nume distincte

$$m:P/(n:Q/R)=n:Q/(m:P/R)$$

Utilizarea recursivității în definirea proceselor subordonate este suficient de surprinzătoare pentru a se ivi dubii dacă într-adevăr funcționează corect. Aceste dubii pot fi încet-încet eliminate, arătând cum progresează construcția. Exemplul de mai jos folosește urma particulară a comportării unui proces și arată cum se produce acea urmă. Mai important, se arată cum nu pot fi produse alte urme ușor diferite.

Exemplu

X1 O urmă tipică a lui *MULTIME* este

$$s=\langle \text{stânga.1}, \text{dreapta.NU}, \text{stânga.2}, \text{dreapta.NU} \rangle$$

Valoarea lui *MULTIME/s* poate fi calculată într-o serie de pași, folosind L1 și L2

$$\begin{aligned} & \text{MULTIME}/\langle \text{stânga.1} \rangle = \text{dreapta!NU} \rightarrow (\text{rest: MULTIME/BUCLĂ}(1)) \\ \text{astfel că } & \text{MULTIME}/\langle \text{stânga.1}, \text{dreapta.NU} \rangle = (\text{rest: MULTIME/BUCLĂ}(1)) \\ \text{și } & \text{MULTIME}/\langle \text{stânga.1}, \text{dreapta.NU}, \text{stânga.2} \rangle \\ & = (\text{rest: MULTIME}/(\text{rest: stânga!2} \rightarrow \text{rest: dreapta?z} \rightarrow \text{dreapta!z} \rightarrow \text{BUCLĂ}(1))) \\ & = (\text{rest:}(\text{dreapta!NU} \rightarrow (\text{rest: MULTIME/BUCLĂ}(2)))) \\ & \quad //(\text{rest: dreapta?z} \rightarrow \text{dreapta!z} \rightarrow \text{BUCLĂ}(1))) \\ & = \text{rest:}(\text{rest: MULTIME/BUCLĂ}(2))//(\text{dreapta!NU} \rightarrow \text{BUCLĂ}(1)) \end{aligned}$$

$$\text{De aceea } \text{MULTIME}/s = \text{rest:}(\text{rest: MULTIME/BUCLĂ}(2))//\text{BUCLĂ}(1)$$

Este evident din cele de mai sus că

$$\langle \text{stânga.1}, \text{dreapta.NU}, \text{stânga.2}, \text{dreapta.DA} \rangle$$

nu este o urmă a lui *MULTIME*.

Cititorul poate verifica că

$$\begin{aligned} & \text{MULTIME}/s^{\langle \text{stânga.2}, \text{dreapta.DA} \rangle} = \text{MULTIME}/s \\ \text{și } & \text{MULTIME}/s^{\langle \text{stânga.5}, \text{dreapta.NU} \rangle} = \text{rest:}(\text{rest:}(\text{rest: MULTIME} \\ & \quad //\text{BUCLĂ}(5))//\text{BUCLĂ}(2))//\text{BUCLĂ}(1) \end{aligned}$$

□

4.5.2 Diagrame de conexiune

Un proces subordonat poate fi reprezentat în interiorul unui dreptunghi reprezentând procesul care îl utilizează așa cum se arată în 4.5 X1 fig. 4.10.

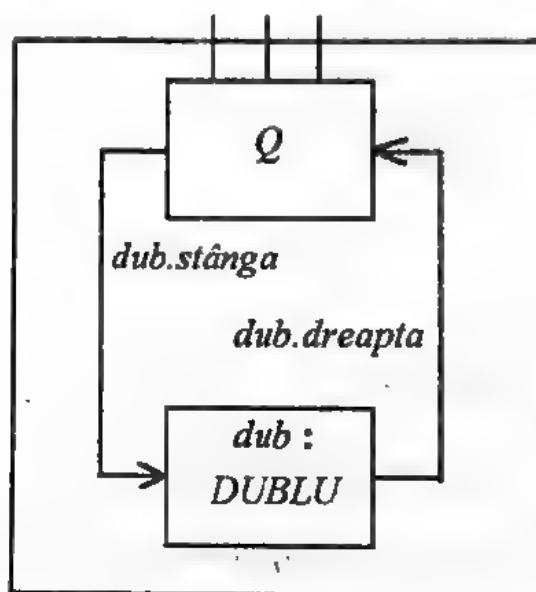


Figura 4.10

Pentru procesele subordonate imbricate, dreptunghiurile se intersectează mai mult, așa cum decurge din 4.5 X2 și se arată în fig. 4.11.

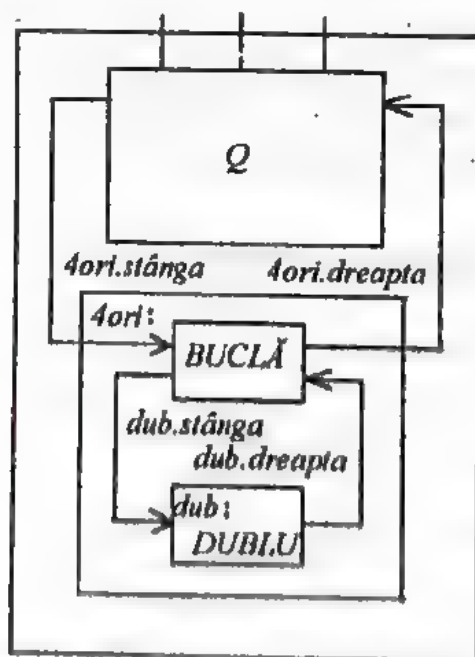
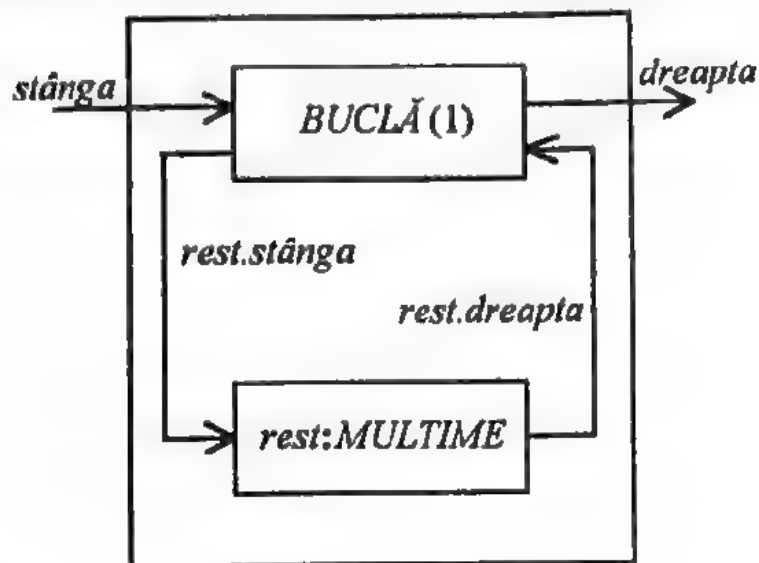


Figura 4.11

$$MULTIME / (st\grave{a}nga1, dreapta.NU) =$$


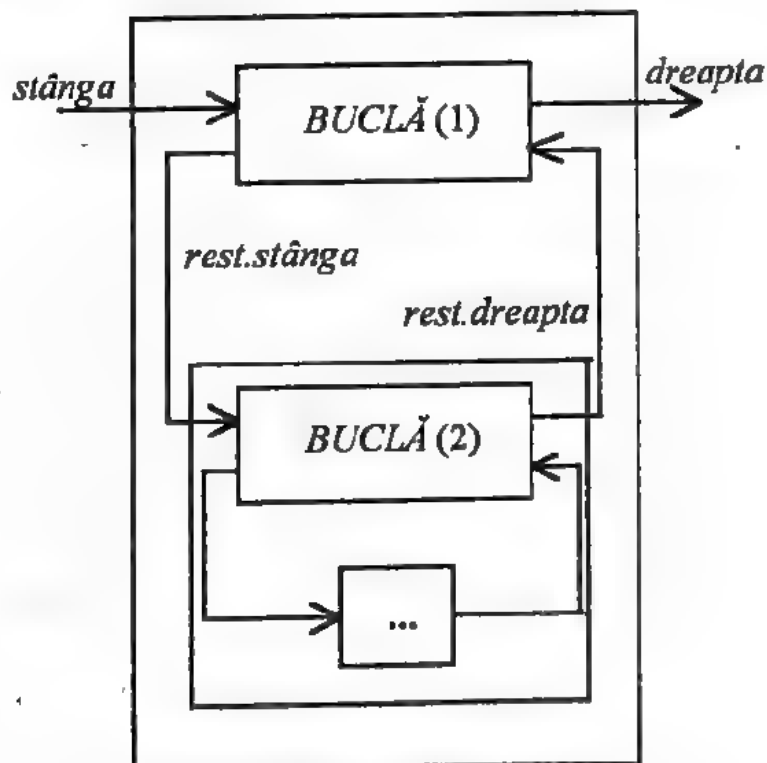
$$MULTIME / s =$$


Figura 4.12

Un proces recursiv este unul imbricat cu sine însuși ca imaginea studioului unui artist, în care există pe un șevalet ultima sa pictură terminată, care arată un șevalet cu ultima sa pictură terminată.... etc. O astfel de pictură complexă în practică nu poate fi realizată. Din fericire pentru un proces, nu este necesar să fie de la început complet – creșterea se produce automat după necesitățile din timpul activității. Astfel (vezi 4.5.1 X1) putem reprezenta etapele succesive din istoria timpurie a unei mulțimi ca în fig. 4.12.

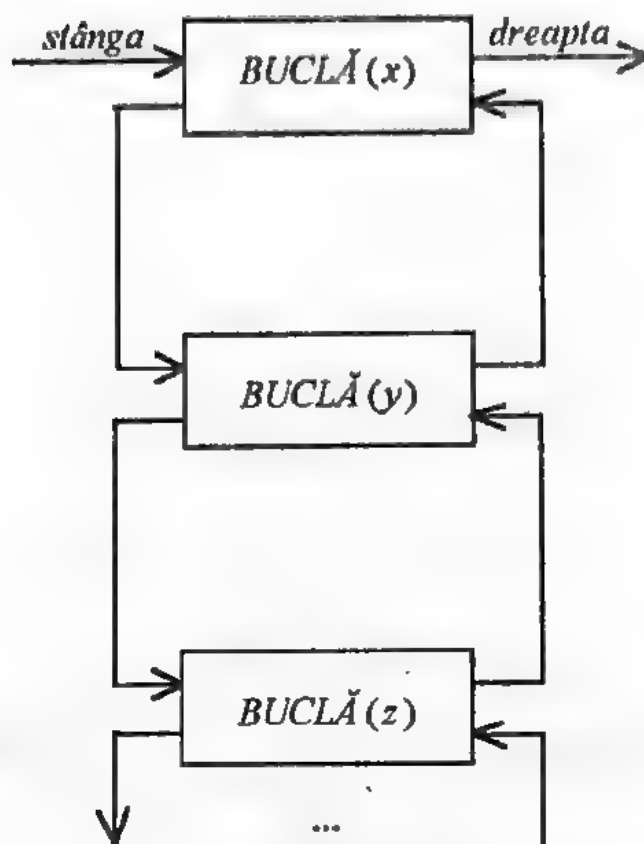


Figura 4.13

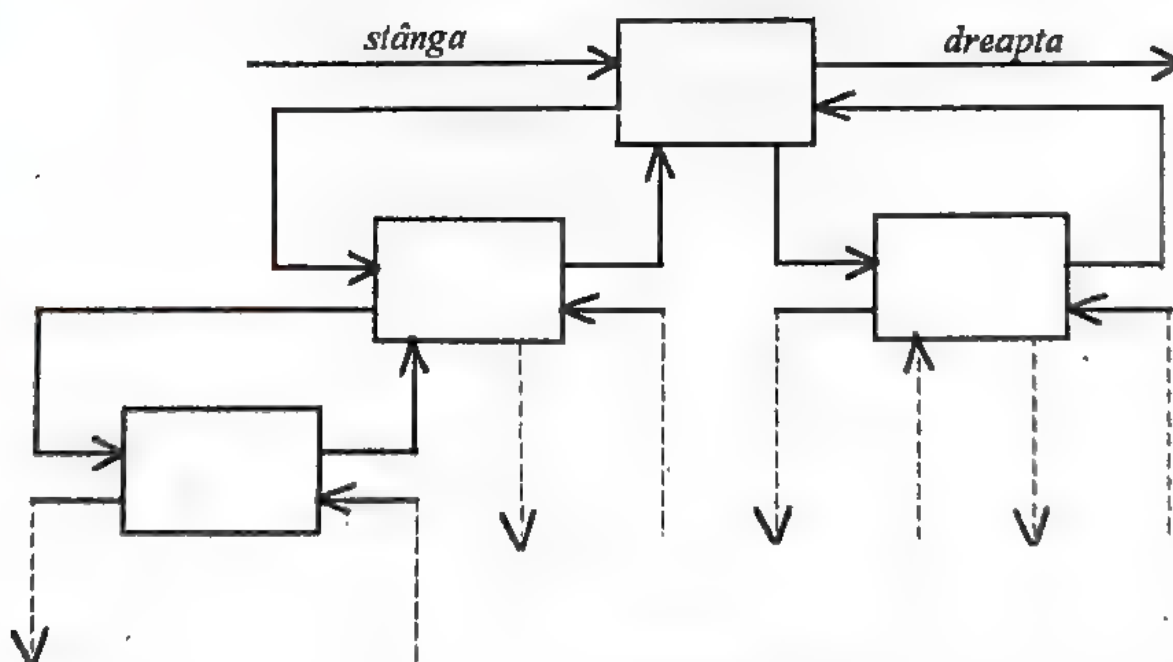


Figura 4.14

Dacă ignorăm imbricarea dreptunghiurilor, atunci putem desena o structură liniară ca în fig. 4.13. Similar, exemplul *ARBORE* (4.5 X9) ar putea fi desenat ca în fig. 4.14.

Diagramele de conexiune de mai sus sugerează cum poate fi construită o rețea de procesare corespunzătoare din componentele hardware, dreptunghiurile reprezentând circuite integrate iar arcele reprezentând legăturile fizice dintre ele. Desigur, în orice realizare practică, recursivitatea trebuie limitată (desfășurată) până la o anumită limită finită pentru ca rețeaua fizică să opereze normal. Dacă această limită este depășită în timpul prelucrărilor, rețeaua nu mai poate funcționa cu succes. Realocarea și reconfigurarea dinamică a rețelelor hardware este mult mai dificilă decât alocarea memoriei bazată pe stivă care face recursivitatea atât de eficientă în programele secvențiale convenționale. Cu toate acestea, în general recursivitatea este desigur justificată prin suportul pe care îl dă în inventarea și proiectarea algoritmilor iar dacă aceasta nu este totul, atunci cel puțin prin bucuria intelectuală pe care o dă acelor care o înțeleg și o utilizează.

5 Procese secvențiale

5.1 Introducere

Procesul *STOP* este prin definiție procesul care nu se angajează în nici o acțiune. Nu este un proces util și probabil rezultă dintr-un blocaj sau altă eroare de proiectare, mai degrabă decât dintr-o alegere deliberată a proiectantului. Totuși există un motiv ca un proces să nu se mai angajeze în nici o acțiune și anume când a terminat de făcut ce avea de făcut. Un astfel de proces se spune că se termină cu succes. Pentru a distinge între acest proces și *STOP* este convenabil să privim terminarea cu succes ca un eveniment special, notat cu simbolul \checkmark (pronunțat "succes"). Un proces secvențial este definit ca unul care are evenimentul \checkmark în alfabetul său și natural acesta este ultimul eveniment în care procesul se poate angaja. Impunem ca \checkmark să nu poată fi o alternativă într-o alegere

$(x.B \rightarrow P(x))$ este invalidă pentru $\checkmark \in B$

$SKIP_A$ este definit ca un proces care nu face nimic dar se termină cu succes.

$$\alpha SKIP_A = A \cup \{\checkmark\}$$

Ca de obicei vom omite frecvent alfabetul subînțeles.

Exemple

X1 Un automat care se intenționează să servească un client numai cu ciocolată sau bomboane și apoi să-și termine activitatea cu succes

$$UNAV = (mon \rightarrow (choc \rightarrow SKIP \mid bonbon \rightarrow SKIP)) \quad \square$$

În proiectarea unui proces care să rezolve un task complex este totdeauna util să divizăm taskul în două subtaskuri, unul din ele trebuind să se termine

cu succes înainte să înceapă un altul. Dacă P și Q sunt procese secvențiale cu același alfabet, compunerea lor secvențială o notăm

$$P;Q$$

fiind un proces care întâi se comportă ca P , dar când P se termină cu succes $(P;Q)$ continuă comportându-se ca Q . Dacă P nu se termină niciodată cu succes nu o face nici $(P;Q)$.

X2 Un automat proiectat să servească doi clienți, unul după altul

$$AVDOI = UNAV; UNAV$$

□

Un proces care repetă aceleași acțiuni atât cât este necesar este cunoscut ca o buclă. El poate fi definit drept un caz special de recursivitate

$$\begin{aligned} *P &= \mu X. (P; X) \\ &= P; P; P; \dots \\ \alpha(*P) &= \alpha P - \{\sqrt{}\} \end{aligned}$$

În mod clar, o astfel de buclă nu se va termina niciodată cu succes. De aceea este util să scoatem $\sqrt{}$ din alfabetul său.

X3 Un automat proiectat să servească orice număr de clienți

$$AVCL = *UNAV$$

Acesta este identic cu $AVCL$ (1.1.3 X3).

□

O secvență de simboluri se spune că este o *propoziție* a unui proces P dacă P se termină cu succes după angajarea în secvența corespunzătoare de acțiuni. Mulțimea tuturor propozițiilor se numește *limbajul* acceptat de P . Astfel notațiile introduse pentru descrierea proceselor secvențiale pot fi folosite de asemenea pentru a defini gramatica unui limbaj simplu, astfel că ar putea deveni utile pentru comunicația între o ființă umană și un calculator.

X4 O propoziție din Pidgingol constă dintr-o clauză substantiv urmată de un predicat. Un predicat este un verb urmat de o clauză substantiv. Verbul este ori *mușcă* ori *zgârie*. Definiția unei clauze substantiv este dată formal mai jos

$$\begin{aligned} \alpha PIDGINGOL &= \{un, o, pisică, câine, mușcă, zgârie\} \\ PIDGINGOL &= CLAUZĂ_SUBSTANTIV; PREDICAT \\ PREDICAT &= VERB; CLAUZĂ_SUBSTANTIV \end{aligned}$$

$VERB = (\text{mușcă} \rightarrow SKIP \mid \text{zgârle} \rightarrow SKIP)$
 $CLAUZĂ_SUBSTANTIV = ARTICOL; SUBSTANTIV$
 $ARTICOL = (\text{un} \rightarrow SKIP \mid \text{o} \rightarrow SKIP)$
 $SUBSTANTIV = (\text{pisică} \rightarrow SKIP \mid \text{câine} \rightarrow SKIP)$

Exemple de propoziții din Pidgingol sunt

o pisică zgârle un câine
un câine mușcă o pisică

□

Pentru a descrie limbajele cu un număr infinit de propoziții este necesar să folosim sau iterația sau recursivitatea.

X5 O clauză adjectiv poate conține orice număr de adjective *mare* sau *cuminte*.

$CLAUZĂ_SUBSTANTIV = ARTICOL; \mu X. (\text{mare} \rightarrow X \mid \text{cuminte} \rightarrow X$
 $\mid \text{pisică} \rightarrow SKIP \mid \text{câine} \rightarrow SKIP)$

Exemple de clauze substantiv sunt

un mare mare cuminte câine
o pisică

□

X6 Un proces care acceptă orice număr de *a*-uri urmate de *b*-uri și apoi același număr de *c*-uri după care se termină cu succes

$A^n B C^n = \mu X. (b \rightarrow SKIP \mid a \rightarrow (X; (c \rightarrow SKIP)))$

Dacă întâi este acceptat un *b*, procesul se termină fără ca nici un *a* sau *c* să nu fie acceptat astfel că numărul lor este același. Dacă este parcursă a doua ramură, propoziția acceptată pornește cu *a* și se termină cu *c* și între aceste două evenimente este propoziția acceptată de apelul recursiv pe procesul *X*. Dacă presupunem că apelul recursiv acceptă un număr egal de *a*-uri și *c*-uri așa va face și apelul nerecursiv pe $A^n B C^n$ deoarece acceptă un *a* în plus la început și un *c* în plus la sfârșit.

Acest exemplu ne arată cum o compunere secvențială folosită în conjuncție cu recursivitatea poate defini o mașină cu un număr infinit de stări.

□

X7 Un proces care întâi se comportă ca $A^n B C^n$ dar apoi acceptă un *d* urmat de același număr de *e*-uri

$$A^n B C^n D E^n = ((A^n B C^n); d \rightarrow \text{SKIP} \parallel C^n D E^n$$

$$\text{unde } C^n D E^n = F(A^n B C^n) \quad \text{cu } \begin{aligned} f(a) &= c \\ f(b) &= d \\ f(c) &= e \end{aligned}$$

— În acest exemplu, procesul din stânga lui \parallel este responsabil pentru asigurarea unui număr egal de a -uri și c -uri (separat de un b). Nu se va permite astfel un d până când nu a sosit același număr de c -uri iar e -urile care nu sunt în alfabetul său sunt ignorate. Procesul din dreapta lui \parallel este responsabil pentru asigurarea unui număr egal de e -uri și c -uri. El ignoră a -urile și b -ul care nu sunt în alfabetul său. Perechea de procese se termină împreună când ambele și-au completat sarcinile repartizate. \square

Notățiile pentru definirea unui limbaj cu ajutorul unui proces sunt tot atât de puternice ca și expresiile regulate. Utilizarea recursivității aduce ceva din puterea gramaticilor libere de context dar nu numai. Un proces poate defini numai acele limbaje care pot fi analizate gramatical de la stânga la dreapta fără *backtracking* sau *look-ahead*. Aceasta din cauză că utilizarea operatorului alegere impune ca primul eveniment al unei alternative să fie diferit de primul eveniment al oricărei alteia. Mai mult, nu este posibil să utilizăm construcția din X5 pentru a defini o clauză substantiv în care cuvântul *cuminte* poate fi atât un substantiv cât și un adjectiv sau amândouă, de exemplu *un cuminte câine*, *un mare cuminte*. Utilizarea lui \square (paragraful 3.3) nu ne-ar ajuta din cauză că introduce nedeterminismul și permite alegerea arbitrară a clauzei care va analiza restul intrării. Dacă alegerea este greșită, procesul se va bloca înaintea atingerii sfârșitului textului de intrare. Ceea ce este deci necesar pentru a rezolva această problemă este un nou tip de operator alternativă care să asigure nedeterminismul angelic ca de exemplu *sau3* (paragraful 3.2.2). Acest nou operator impune ca două alternative să ruleze concurent până când mediul va face alegerea. Definiția sa este lăsată ca exercițiu.

Fără nedeterminismul angelic metoda definirii limbajului descrisă mai sus nu este tot atât de puternică ca și gramaticile libere de context din cauză că ea necesită analiză gramaticală stânga-dreapta fără *backtracking*. Totuși, introducerea lui \parallel permite definirea limbajelor care nu sunt libere de context pentru exemplul X7.

X8 Un proces care acceptă orice întrețesere de *jos* și *sus* cu excepția terminării cu succes cu prima ocazie când numărul de evenimente *jos* depășește numărul de evenimente *sus*

$$POZ = (jos \rightarrow \text{SKIP} \mid sus \rightarrow (POZ; POZ))$$

Dacă primul simbol este *jos* procesul *POZ* este imediat terminat. Dar dacă primul simbol este *sus*, este necesar să se accepte încă două evenimente *jos* față de *sus*. Singura cale de a realiza aceasta este acceptarea mai întâi a unui *jos* în plus față de evenimente *sus*, ulterior acceptându-se din nou un *jos* în plus față de un *sus*. De aceea sunt necesare două apeluri succesive recursive ale lui *POZ*, unul după altul. \square

X9 Procesul M_0 se comportă ca MM_0 (1.1.4 X2)

$$\begin{aligned} M_0 &= (pe_loc \rightarrow M_0 \mid sus \rightarrow M_1) \\ M_{n+1} &= POZ; M_n \\ &= \underbrace{POZ; \dots POZ; POZ; M_0}_{n \text{ ori}} \end{aligned} \quad \begin{array}{l} \text{pentru toți } n \geq 0 \\ \square \end{array}$$

Acum putem rezolva problema menționată în 2.6.2 X3 și întâlnită din nou în 4.5 X3, astfel că fiecare operație cu un proces subordonat să menționeze explicit restul procesului utilizator care urmează după el. Efectul scontat poate fi acum mult mai convenabil realizat cu ajutorul lui *SKIP* și a compunerii secvențiale.

X10 Un proces *UTIL* manipulează două variabile contor numite *l* și *m* (vezi 2.6.2 X3)

$$l:MM_0 \parallel m:MM_3 \parallel UTIL$$

Următorul proces (în interiorul lui *UTIL*) adună valoarea curentă a lui *l* la *m*

$$ADUN = (l.pe_loc \rightarrow SKIP \mid l.jos \rightarrow (ADUN; (m.sus \rightarrow l.sus \rightarrow SKIP)))$$

Dacă valoarea lui *l* este inițial 0, nu mai trebuie făcut nimic deoarece procesul se termină. Altfel, *l* este decrementat și valoarea actualizată (redușă) este adunată la *m* (printr-un apel recursiv la *ADUN*). Apoi *m* este incrementat încă o dată și *l* este de asemenea incrementat pentru a compensa decrementarea sa inițială fiind astfel adus la valoarea inițială. \square

5.2 Legi

Legile pentru compunerea secvențială sunt similare acelor pentru concatenare (paragraful 1.6.1), *SKIP* jucând rolul de unitate

$$L1 \quad SKIP; P = P; SKIP = P$$

$$L2 \quad (P;Q);R = P;(Q;R)$$

$$L3 \quad (x:B \rightarrow P(x));Q = (x:B \rightarrow (P(x);Q))$$

Legea pentru operatorul alegere are corolarele

$$L4 \quad (a \rightarrow P);Q = a \rightarrow (P;Q)$$

$$L5 \quad STOP;Q = STOP$$

Când procesele secvențiale se compun în paralel, combinația se termină cu succes numai când ambele componente sunt în această situație

$$L6 \quad SKIP_A || SKIP_B = SKIP_{A \cup B}$$

Un proces care se termină cu succes nu mai participă în nici un eveniment oferit de un partener concurent

$$L7 \quad ((x:B \rightarrow P(x)) || SKIP_A) = (x:(B-A) \rightarrow (P(x) || SKIP_A))$$

Ne punem întrebarea dacă o combinație concurentă de proces secvențial cu proces nesecvențial se termină cu succes. Dacă alfabetul procesului secvențial conține în întregime pe cel al partenerului, terminarea colaborării este determinată de procesul secvențial, deoarece celălalt proces nu mai poate face nimic când partenerul său și-a terminat activitatea

$$L8 \quad STOP_A || SKIP_B = SKIP_B \quad \text{dacă } \checkmark \in A \wedge A \subseteq B$$

Condiția pentru validitatea acestei legi este una foarte rezonabilă care ar putea fi totdeauna observată deoarece \checkmark este numai în alfabetul unuia din cele două procese evoluând concurent. Astfel evităm situația ca un proces să continue după ce s-a angajat în \checkmark .

Legile L1-L3 pot fi folosite pentru a demonstra afirmația din 5.1 X9 că M_0 se comportă ca MM_0 (1.1.4 X2). Aceasta se face arătând că M_n satisface mulțimea de ecuații recursive cu gardă folosite pentru a defini MM_n . Ecuația pentru M_0 este aceeași ca pentru MM_0

$$M_0 = (pe_loc \rightarrow M_0 \mid sus \rightarrow M_1) \quad \text{definiție } M_0$$

Pentru $n > 0$, trebuie să dovedim că

$$M_n = (sus \rightarrow M_{n+1} \mid jos \rightarrow M_{n-1})$$

Demonstrație

$$\begin{aligned}
 MS &= POZ; M_{n-1} \\
 &= (jos \rightarrow SKIP \mid sus \rightarrow POZ; POZ); M_{n-1} \\
 &= (jos \rightarrow (SKIP; M_{n-1}) \mid sus \rightarrow (POZ; POZ); M_{n-1}) \\
 &= (jos \rightarrow M_{n-1} \mid sus \rightarrow POZ; (POZ; M_{n-1})) \\
 &= (jos \rightarrow M_{n-1} \mid sus \rightarrow POZ; M_n) \\
 &= MD
 \end{aligned}$$

definiție M_n
 definiție POZ
 L3
 L1, L2
 definiție M_n
 definiție M_n

Deoarece M_n se supune la aceeași mulțime de ecuații recursive cu gardă ca și MM_n , procesele sunt aceleași.

Această demonstrație a fost prezentată complet pentru a ilustra utilizarea legilor și a elimina suspiciunile provocate de transformările recurente. Ceea ce pare cel mai curios este că demonstrația nu folosește inducția după n . De fapt, orice încercare de a folosi inducția după n va eșua din cauză că definiția lui MM_n conține procesul MM_{n+1} . Din fericire, aducerea aminte a legii soluției unice ne scoate evident din impas.

5.3 Aspecte matematice

Definiția matematică a compunerii secvențiale trebuie formulată astfel încât să fim siguri de adevărul legilor menționate în paragraful anterior. O atenție specială trebuie exercitată asupra

$$P; SKIP = P$$

Ca de obicei, tratarea proceselor deterministe este mult mai simplă și se va face mai întâi.

5.3.1 Procese deterministe

Operațiile cu procese deterministe sunt definite cu ajutorul urmelor rezultate-
lor lor. Prima și singura acțiune a procesului $SKIP$ este terminarea cu succes
astfel încât el are numai două urme

$$L0 \quad urme(SKIP) = \{\diamond, \surd\}$$

Pentru a defini compunerea secvențială a proceselor este convenabil să definim
mai întâi compunerea secvențială a urmelor lor individuale. Dacă s și t sunt
urme și s nu conține \surd

$$\begin{aligned}
 \langle s, \diamond \rangle &= s \\
 \langle s \surd; t \rangle &= t
 \end{aligned}$$

(vezi paragraful 1.9.7. pentru detalii complete). O urmă a lui $(P;Q)$ constă dintr-o urmă a lui P şi dacă aceasta se termină cu \checkmark , evenimentul este înlocuit şi se permite generarea urmei lui Q

$$L1 \quad \text{urme}(P;Q) = \{s; t \mid s \in \text{urme}(P) \wedge t \in \text{urme}(Q)\}$$

O definiţie echivalentă este

$$L1A \quad \text{urme}(P;Q) = \{s \mid s \in \text{urme}(P) \wedge \neg \langle \checkmark \rangle \text{ în } s\} \\ \cup \{s \wedge t \mid s \wedge \langle \checkmark \rangle \in \text{urme}(P) \wedge t \in \text{urme}(Q)\}$$

Această definiţie este mai simplu de înţeles decât de folosit.

Scopul simbolului \checkmark este acela de a permite terminarea procesului care se angajează în el. De aceea avem nevoie de legea

$$L2 \quad P/s = \text{SKIP} \quad \text{dacă } s \wedge \langle \checkmark \rangle \in \text{urme}(P)$$

Această lege este esenţială în demonstrarea lui

$$P; \text{SKIP} = P$$

Din nefericire, ea nu este în general adevărată. De exemplu, dacă

$$P = (\text{SKIP}_{\{c\}} \parallel c \rightarrow \text{STOP}_{\{c\}})$$

atunci $\text{urme}(P) = \{ \langle \rangle, \langle \checkmark \rangle, \langle c \rangle, \langle c, \checkmark \rangle, \langle \checkmark, c \rangle \}$ şi $P/\langle \rangle \neq \text{SKIP}$ cu toate că $\langle \checkmark \rangle \in \text{urme}(P)$. De aceea avem nevoie să impunem nişte constrângeri de alfabet pentru compunerea paralelă. Construcţia $(P \parallel Q)$ trebuie privită ca invalidă cu excepţia situaţiei

$$\alpha P \subseteq \alpha Q \vee \alpha Q \subseteq \alpha P \vee \checkmark \in (\alpha P \cap \alpha Q \cup \overline{\alpha P} \cap \overline{\alpha Q})$$

Pentru motive similare schimbarea de alfabet trebuie garantată că păstrează \checkmark nealterat, astfel că $f(P)$ este invalidă fără

$$f(\checkmark) = \checkmark$$

Mai departe, dacă m este un nume de proces trebuie să adoptăm convenţia ca

$$m.\checkmark = \checkmark$$

În fine, nu trebuie să folosim alegerea în situaţia

$$(\sqrt{\rightarrow}P | c \rightarrow Q)$$

Această restricție se aplică și lui RUN_A când $\sqrt{\in A}$.

5.3.2 Procese nedeterministe

Compunerea secvențială a proceselor nedeterministe prezintă un număr de probleme. Prima este că un proces nedeterminist ca

$$SKIP \sqcap (c \rightarrow SKIP)$$

nu satisface legea L2 din paragraful anterior. O soluție ar fi să slăbim legea 5.3.1. L2 la

$$L2A \quad s \prec \sqrt{\in} urme(P) \Rightarrow (P/s) \subseteq SKIP$$

Aceasta înseamnă că ori de câte ori P se termină, o poate face fără a oferi vreun eveniment ca alternativă mediului. Pentru a menține adevărul din L2A trebuie observate toate restricțiile din paragraful anterior astfel că

$SKIP$ nu trebuie să apară niciodată fără gardă ca operand în \square
 $\sqrt{}$ nu trebuie să apară în alfabetul vreunui operand din \parallel

(Este posibil ca o ușoară modificare în definiția lui \square sau \parallel să permită relaxarea acestor restricții).

În plus față de legile date mai înainte în acest capitol, compunerea secvențială a proceselor nedeterministe va satisface următoarele legi. Mai întâi, un proces divergent rămâne divergent fără a ține seama de ce este specificat să se întâmple după terminarea lui cu succes.

$$L1 \quad CHAOS;P = CHAOS$$

Compunerea secvențială se distribuie cu alternativa nedeterministă

$$L2A \quad (P \sqcap Q);R = (P;R) \sqcap (Q;R)$$

$$L2B \quad R;(P \sqcap Q) = (R;P) \sqcap (R;Q)$$

Pentru a defini $(P;Q)$ în modelul matematic al proceselor nedeterministe (paragraful 3.9), este necesară tratarea eșecurilor și divergențelor. Dar mai întâi să descriem refuzurile (paragraful 3.4). Dacă P poate refuza X și nu se poate termina cu succes, urmează că $X \cup \{\sqrt{}\}$ este de asemenea un refuz al lui P (3.4.1 L11). În acest caz, X este un refuz al lui $(P;Q)$. Dar dacă P oferă opțiunea terminării cu succes atunci în $(P;Q)$ această acțiune poate apare indepen-

dent. Producerea ei este mascată și orice refuz al lui Q este de asemenea un refuz al lui $(P;Q)$. Cazul când terminarea cu succes al lui P este nedeterministă este de asemenea inclus în această definiție

$$D1 \quad \text{refuzuri}(P;Q) = \{X \mid (X \cup \{\checkmark\}) \in \text{refuzuri}(P)\} \\ \cup \{X \mid \langle \checkmark \rangle \in \text{urme}(P) \wedge X \in \text{refuzuri}(Q)\}$$

Urmele lui $(P;Q)$ sunt definite în aceeași manieră ca pentru procesele deterministe. Divergențele lui $(P;Q)$ sunt definite cu observația că ansamblul diverge ori de câte ori P diverge sau când P s-a terminat cu succes și apoi diverge Q .

$$D2 \quad \text{divergențe}(P;Q) = \{s \mid s \in \text{divergențe}(P) \wedge \neg \langle \checkmark \rangle \text{ in } s\} \\ \cup \{s \wedge t \mid s \wedge \checkmark \in \text{urme}(P) \wedge \neg \langle \checkmark \rangle \text{ in } s \wedge t \in \text{divergențe}(Q)\}$$

Orice eșec al lui $(P;Q)$ este ori un eșec al lui P înainte ca P să se poată termina, ori un eșec al lui Q după ce P s-a terminat cu succes

$$D3 \quad \text{eșecuri}(P;Q) = \{(s, X) \mid (s, X \cup \{\checkmark\}) \in \text{eșecuri}(P)\} \\ \cup \{(s \wedge t, X) \mid s \wedge \checkmark \in \text{urme}(P) \wedge (t, X) \in \text{eșecuri}(Q)\} \cup \{(s, X) \mid s \in \text{divergențe}(P;Q)\}$$

5.3.3 Implementare

SKIP este implementat ca un proces care acceptă numai simbolul "SUCCESS. Nu contează ce face după aceea.

$$SKIP = \lambda x. \text{if } x = \text{"SUCCESS"} \text{ then } STOP \text{ else "BLIP"}$$

O compunere secvențială se comportă ca al doilea operand dacă primul operand se termină, altfel primul operand participă în primul eveniment și restul din el participă la compunerea cu al doilea operand.

$$\text{secvență}(P, Q) = \text{if } P(\text{"SUCCESS"}) \neq \text{"BLIP"} \text{ then } Q \\ \text{else } \lambda x. \text{if } P(x) = \text{"BLIP"} \text{ then "BLIP"} \\ \text{else } \text{secvență}(P(x), Q)$$

5.4 Întreruperi

În acest paragraf definim o formă de compunere secvențială $(P \wedge Q)$ care nu depinde de terminarea cu succes a lui P . Mai mult, progresul lui P este întrerupt de apariția primului eveniment din Q , iar P nu mai este niciodată reluat.

Urmează că o urmă a lui $(P \wedge Q)$ este chiar o urmă a lui P până la un punct, când apare întreruperea, urmată de orice urmă a lui Q

$$\alpha(P \wedge Q) = \alpha P \cup \alpha Q$$

$$\text{urme}(P \wedge Q) = \{s \wedge t \mid s \in \text{urme}(P) \wedge t \in \text{urme}(Q)\}$$

Pentru a evita problemele precizăm că $\sqrt{}$ nu trebuie să fie în αP .

Următoarea lege afirmă că mediul este cel care determină când va începe Q prin selecția unui eveniment care este oferit inițial de Q dar nu de P

$$L1 \quad (x:B \rightarrow P(x)) \wedge Q = Q \sqcap (x:B \rightarrow (P(x) \wedge Q))$$

Dacă $(P \wedge Q)$ poate fi întrerupt de R aceasta este totuna cu P să poată fi întrerupt de $(Q \wedge R)$

$$L2 \quad (P \wedge Q) \wedge R = P \wedge (Q \wedge R)$$

Deoarece *STOP* nu se poate angaja în nici un eveniment nu poate fi declanșat de mediu. Similar, dacă *STOP* este interruptibil, de fapt numai întreruperea poate să apară. Astfel, *STOP* este unitatea lui \wedge

$$L3 \quad P \wedge \text{STOP} = P = \text{STOP} \wedge P$$

Operatorul întrerupere execută amândoi operanzii săi cel mult o dată, astfel că se distribuie cu alternativa nedeterministă.

$$L4A \quad P \wedge (Q \sqcap R) = (P \wedge Q) \sqcap (P \wedge R)$$

$$L4B \quad (Q \sqcap R) \wedge P = (Q \wedge P) \sqcap (R \wedge P)$$

În fine, nimeni nu poate schimba natura unui proces divergent prin întrerupere, nici nu este prea sigură specificarea unui proces divergent după întrerupere

$$L5 \quad \text{CHAOS} \wedge P = P \wedge \text{CHAOS} = \text{CHAOS}$$

În restul acestui paragraf vom insista pe faptul că evenimentele inițiale posibile ale proceselor care forțează întreruperea nu există în alfabetul procesului întrerupt. Deoarece apariția întreruperii este vizibilă și controlabilă de mediu, această restricție păstrează determinismul și efectele asupra operatorilor sunt simplificate. Pentru a evidenția conservarea determinismului extindem definiția operatorului alegere. În condițiile când $c \notin B$

$(x:B \rightarrow P(x) \mid c \rightarrow Q)$ este prescurtarea pentru

$$(x:(B \cup \{c\}) \rightarrow (\text{if } x=c \text{ then } Q \text{ else } P(x)))$$

și similar pentru mai mulți operanzi.

5.4.1 Catastrofa

Fie K simbolul pentru un eveniment de întrerupere catastrofic, pe care este rezonabil să-l presupunem că n-ar fi cauzat de P . Mai formal

$$K \notin \alpha P$$

Atunci un proces care se comportă ca P până la catastrofă și apoi ca Q este definit

$$P \hat{K} Q = P \wedge (K \rightarrow Q)$$

Aici Q este un proces cu posibilitatea de a se reface după catastrofă. De notat că operatorul \hat{K} este deosebit de evenimentul K . Prima lege este chiar o formalizare evidentă a descrierii operatorului

$$L1 \quad (P \hat{K} Q)(s \prec K) = Q \quad \text{pentru } s \in \text{urme}(P)$$

În modelul determinist, această singură lege definește unic înțelesul operatorului. Într-un univers nedeterminist, unicitatea ar necesita legi adiționale afirmând strictețea și distributivitatea în ambele argumente.

A doua lege dă o descriere mai explicită a primului și următorilor pași ai procesului, arătându-ne cum se distribuie \hat{K} înapoi cu \rightarrow

$$L2 \quad (x:B \rightarrow P(x)) \hat{K} Q = (x:B \rightarrow (P(x) \hat{K} Q) \mid K \rightarrow Q)$$

Această lege definește în mod unic operatorul în procesele deterministe.

5.4.2 Restart

Un posibil răspuns la catastrofă este reluarea procesului original. Fie P un proces astfel ca $K \notin \alpha P$. Specificăm \hat{P} ca un proces care se comportă ca P până ce apare K și după fiecare eveniment K se comportă ca P de la început. Un astfel de proces este numit *restartabil* și este definit de recursivitatea simplă

$$\alpha \hat{P} = \alpha P \cup \{K\}$$

$$\begin{aligned} \hat{P} &= \mu X. (P \hat{X}) \\ &= P \hat{X} (P \hat{X} (P \hat{X} \dots)) \end{aligned}$$

Aceasta este o recursivitate cu gardă deoarece apariția lui X este cu gardă datorită lui K . Procesul \hat{P} este desigur un proces ciclic (paragraful 1.8.3) chiar dacă P nu este. Catastrofa nu este singurul motiv pentru restart. Considerăm un proces proiectat să prezinte un joc interacționând cu utilizatorul uman prin intermediul selecției tastelor unei tastaturi (vezi descrierea funcției *interact* din paragraful 1.4). Oamenii se plictisesc uneori ușor de un joc și vor să înceapă un altul. Pentru acest motiv este prevăzută o tastă nouă, specială, pe tastatură. Apăsarea ei în orice moment al jocului va restarta jocul. Este convenabil să definim un joc P independent de facilitatea de restartare și apoi să-l transformăm într-un joc \hat{P} utilizând operatorul de mai sus. Această idee se datorește lui Alex Teruel.

Definiția formală a lui \hat{P} este exprimată de legea

$$L1 \quad \hat{P} / s \wedge \langle K \rangle = \hat{P} \quad \text{pentru } s \in \text{urme}(P)$$

Dar această lege nu definește unic \hat{P} deoarece ea este satisfăcută și de RUN . Totuși, P este cel mai mic proces determinist care satisface L1.

5.4.3 Alternarea

Presupunem că P și Q sunt procese care joacă jocuri în maniera descrisă în paragraful 5.4.2. Un om dorește să joace ambele jocuri simultan, alternând între ele la fel cum un maestru de șah joacă un simultan ciclând succesiv pe la partenerii mai slabi. De aceea prevedem o nouă tastă \otimes care determină alternarea între cele două jocuri P și Q . Aceasta este similar unei întreruperi prin aceea că jocul curent este întrerupt într-un punct arbitrar, dar diferă de întrerupere prin faptul că starea curentă a jocului curent este salvată astfel că poate fi reluat când celălalt joc este întrerupt la un moment dat. Procesul care descrie jocul P și Q simultan în acest mod este notat $(P \otimes Q)$ și este cel mai clar specificat de legile

$$L1 \quad \otimes \in (\alpha(P \otimes Q) - \alpha P - \alpha Q)$$

$$L2 \quad (P \otimes Q) / s = (P / s) \otimes Q$$

$$L3 \quad (P \otimes Q) / \langle \otimes \rangle = (Q \otimes P)$$

$$\text{dacă } s \in \text{urme}(P)$$

Dorim să aflăm cel mai mic proces care satisface L2 și L3. O descriere mai constructivă a operatorului poate fi derivată din aceste legi, arătându-ne cum \otimes se distribuie înapoi cu \rightarrow

$$L4 \quad (x:B \rightarrow P(x)) \otimes Q = (x:B \rightarrow (P(x) \otimes Q)) \mid \otimes \rightarrow (Q \otimes P)$$

Operatorul alternare este util nu numai pentru jocuri. O facilitare asemănătoare ar putea fi asigurată într-un sistem de operare "prietenos" cu facilități de alternare a utilităților. De exemplu nu dorim să pierdem poziția în editor când apelăm un program "help" și nici viceversa.

5.4.4 Puncte de control

Fie P un proces care descrie comportarea unui sistem de baze de date pe o perioadă lungă de timp. Când apare (\mathcal{H}) una din cele mai dificile situații ar fi ca P să repornească din starea inițială, pierzând toate datele multiple acumulate de sistem. Ar fi mult mai convenabil să ne reîntoarcem la o stare mai recentă a sistemului care se știe că este satisfăcătoare. O astfel de stare este cunoscută ca punct de control (checkpoint). De aceea afectăm o nouă tastă \odot care ar putea fi apăsată numai când starea curentă a sistemului este cunoscută ca fiind satisfăcătoare. Când apare \mathcal{H} , este restaurat cel mai recent punct de control. Dacă nu există nici unul, se restaurează starea inițială. Presupunem că \odot și \mathcal{H} nu sunt în alfabetul lui P și definim $Pc(P)$ un proces care se comportă ca P dar răspunde într-o manieră adecvată acestor două evenimente.

Definiția formală a lui $Pc(P)$ este cel mai pe scurt formalizată de legile

$$\begin{array}{ll} L1 & Pc(P) / s^{\wedge} \langle \mathcal{H} \rangle = Pc(P) \quad \text{pentru } s \in \text{urme}(P) \\ L2 & Pc(P) / s^{\wedge} \langle \odot \rangle = Pc(P/s) \quad \text{pentru } s \in \text{urme}(P) \end{array}$$

$Pc(P)$ poate fi definit mai explicit în termenii unui operator binar $Pc2(P, Q)$, unde P este procesul curent și Q este cel mai recent punct de control așteptând să fie reluat. Dacă catastrofa apare înainte de primul punct de control, sistemul repornește așa cum se descrie în legile

$$\begin{array}{ll} L3 & Pc(P) = Pc2(P, P) \\ L4 & \text{Dacă } P = (x:B \rightarrow P(x)) \\ & \text{atunci } Pc2(P, Q) = (x:B \rightarrow Pc2(P(x), Q)) \mid \mathcal{H} \rightarrow Pc2(Q, Q) \mid \odot \rightarrow Pc2(P, P) \end{array}$$

Legea L4 este sugestivă pentru o metodă de implementare practică în care starea din punctul de control este reținută pe un suport ieftin dar durabil ca de exemplu disc magnetic sau bandă. Când apare evenimentul \odot , starea curentă este copiată ca noul punct de control. Când apare \mathcal{H} , punctul de control este copiat ca noua stare curentă. Pentru motive de economie, un implementa-

tor de sistem se asigură de faptul că datele sunt cât mai mult partajate între starea curentă și stările din punctele de control. O astfel de optimizare este foarte dependentă de mașină și aplicație. Iată pentru ce este așa de plăcută și simplă matematica.

Operatorul de punct de control este util nu numai pentru sisteme baze de date pe scară mare. Când jucăm un joc dificil putem dori să explorăm o anumită direcție de joc fără a intra în ea. Astfel se apasă © pentru a reține poziția curentă și dacă explorările sunt fără succes, folosirea tastei va restabili starea din punctul de control.

Aceste idei despre puncte de control au fost explorate de Ian Hayes.

5.4.5 Puncte de control multiple

Când utilizăm un sistem cu puncte de control $Pc(P)$ se poate întâmpla ca un punct de control să genereze eroare. În astfel de cazuri este de dorit să anulăm cel mai recent punct de control și să mergem la anteriorul. Pentru aceasta avem nevoie de un sistem care să rețină două sau mai multe din cele mai recente stări din puncte de control corespunzătoare. În principiu, nu există nici un motiv pentru care n-am defini un sistem $Pcm(P)$, care reține în timp toate punctele de control de la început. Fiecare apariție a lui \curvearrowright returnează starea ce era chiar *înaintea* celui mai recent © și nu starea de după el. Ca întotdeauna insistăm

$$\alpha Pcm(P) - \alpha P = \{ \textcircled{C}, \curvearrowright \}$$

Un \curvearrowright înaintea unui © ne întoarce la început

$$L1 \quad Pcm(P) s \wedge \curvearrowright = Pcm(P) \quad \text{pentru } s \in urme(P)$$

Un \curvearrowright după un © anulează efectul a tot ce s-a întâmplat până atunci *incluzând* cel mai recent ©.

$$L2 \quad Pcm(P) s \wedge \textcircled{C} \wedge \curvearrowright = Pcm(P) s \quad \text{pentru } (s \upharpoonright \alpha P) \wedge t \in urme(P)$$

O descriere mai explicită a lui $Pcm(P)$ poate fi dată în termenii unui operator binar $Pcm2(P, Q)$, unde P este procesul curent și Q este format dintr-o stivă de puncte de control așteptând să fie reluate dacă e necesar. Conținutul inițial al stivei este o secvență infinită de copii ale lui P

$$\begin{aligned} L3 \quad Pcm(P) &= \mu X. Pcm2(P, X) \\ &= Pcm2(P, Pcm(P)) \\ &= Pcm2(P, Pcm2(P, Pcm2(P, \dots))) \end{aligned}$$

5.5 Asignare

În acest paragraf vom introduce cele mai importante aspecte ale programării secvențiale convenționale, anume asignările, condițiile și buclele. Pentru a simplifica formularea legilor utile vor fi definite câteva notații mai speciale.

Principala caracteristică a programării convenționale este asignarea. Dacă x este o variabilă de program, e este o expresie și P un proces

$$(x:=e; P)$$

este un proces care se comportă ca P cu deosebirea că valoarea inițială a lui x se definește a fi valoarea inițială a expresiei e . Valorile inițiale ale celorlalte variabile sunt neschimbate. Asignarea însăși poate fi definită

$$(x:=e) = (x:=e; \text{SKIP})$$

Asignarea simplă se generalizează ușor la asignare multiplă. Fie x o listă de variabile distincte

$$x = x_0, x_1, \dots, x_{n-1}$$

Fie e o altă listă de expresii

$$e = e_0, e_1, \dots, e_{n-1}$$

În condițiile în care lungimile celor două liste sunt egale

$$x:=e$$

asignează valoarea inițială a lui e_i la x_i pentru toți i . De notat că toți e_i sunt evaluați înaintea oricărei asignări astfel că dacă variabila y apare în expresia g

$$y:=f, z:=g$$

atunci ea este diferită de

$$y, z:=f, g$$

Fie b o expresie a cărei evaluare este logică (*true* sau *false*). Dacă P și Q sunt procese

$$P \nabla b \nabla Q \quad . \quad (P \text{ dacă } b \text{ altfel } Q)$$

este un proces care se comportă ca P dacă valoarea inițială a lui b este *true* sau ca Q dacă valoarea inițială a lui b este *false*. Notăția este imediată dar mai puțin stânjenitoare ca cea tradițională

if b then P else Q

Pentru motive similare bucla tradițională

while b do Q
va fi scrisă

$b * Q$

Aceasta poate fi definită prin recursivitate

D1 $b * Q = \mu X. ((Q; X) \nrightarrow b \nrightarrow SKIP)$

Exemple

X1 Un proces care se comportă ca MM_n (1.1.4 X2)

$$X1 = \mu X. (pe_loc \rightarrow X \mid sus \rightarrow (n := 1; X)) \\ \nrightarrow n = 0 \nrightarrow \\ (sus \rightarrow (n := n + 1; X) \mid jos \rightarrow (n := n - 1; X))$$

Valoarea curentă a contorului este înregistrată în variabila n □

X2 Un proces care se comportă ca MM_0

$n := 0; X1$

Valoarea inițială a contorului este pusă pe zero □

X3 Un proces care se comportă ca POZ (5.1 X8)

$n := 1; (n > 0) * (sus \rightarrow n := n + 1 \mid jos \rightarrow n := n - 1)$

Recursivitatea a fost înlocuită cu o buclă convențională □

X4 Un proces care divide numărul natural x la numărul natural y , asignând câtul la q și restul la r

$CAT = (q := x \div y; r := x - q \times y)$ □

X5 Un proces cu același efect ca și X4 care calculează câtul prin metoda lentă a scăderii repetate

$$C\hat{A}TLUNG = (q:=0; r:=x; ((r \geq y) * (q:=q+1; r:=r-y)))$$

□

În exemplul anterior (4.5 X3) am arătat cum comportarea unei variabile poate fi modelată printr-un proces subordonat care-și comunică valoarea procesului care-l folosește. În acest capitol am lăsat special la o parte această tehnică pentru că nu are proprietățile matematice pe care le-am dori. De exemplu, dorim

$$(m:=1; m:=1) = (m:=1)$$

dar din nefericire

$$(m.st\acute{a}nga!1 \rightarrow m.st\acute{a}nga!1 \rightarrow SKIP) \neq (m.st\acute{a}nga!1 \rightarrow SKIP)$$

5.5.1 Legi

În legile de asignare, x și y semnifică liste de variabile distincte. În continuare, $e, f(x), f(e)$ semnifică liste de expresii ce conțin posibile apariții ale variabilelor din listele x sau y , iar $f(e)$ conține e_i dacă $f(x)$ conține x_i pentru toți indicii i . Pentru simplitate vom presupune că toate expresiile dau întotdeauna un rezultat pentru orice valoare a variabilelor pe care le conțin

- L1** $(x:=x) = SKIP$
- L2** $(x:=e; x:=f(x)) = (x:=f(e))$
- L3** dacă x și y este o listă de variabile distincte $(x:=e) = (x,y:=e,y)$
- L4** dacă x,y,z sunt de aceeași lungime ca e,f,g , respectiv
 $(x,y,z:=e,f,g) = (x,z,y:=e,g,f)$

Folosind aceste legi este posibil de transformat orice secvență de asignări într-una singură, o listă a tuturor variabilelor implicate.

Dacă $\langle b \rangle$ este considerat un operator binar infix el posedă câteva proprietăți algebrice familiare

- L5-6** $\langle b \rangle$ este idempotent, asociativ și se distribuie cu $\langle c \rangle$
- L7** $P \langle true \rangle Q = P$
- L8** $P \langle false \rangle Q = Q$
- L9** $P \langle \neg b \rangle Q = Q \langle b \rangle P$
- L10** $P \langle b \rangle (Q \langle b \rangle R) = P \langle b \rangle R$
- L11** $P \langle a \langle b \rangle c \rangle Q = (P \langle a \rangle Q) \langle b \rangle (P \langle c \rangle Q)$

$$\text{L12 } x:=e; (P \nabla b(x) \nabla Q) = (x:=e; P) \nabla b(e) \nabla (x:=e; Q)$$

$$\text{L13 } (P \nabla b \nabla Q); R = (P; R) \nabla b \nabla (Q; R)$$

Pentru a testa efectiv asignarea în procese concurente este necesar să impunem restricția ca nici o variabilă asignată într-un proces concurent să nu fie folosită în altul. Pentru a întări această restricție introducem două noi categorii de simboluri în alfabetele proceselor secvențiale.

$\text{var}(P)$ mulțimea variabilelor care pot fi asignate în interiorul lui P
 $\text{acc}(P)$ mulțimea variabilelor care pot fi accesate în expresii din P

Toate variabilele care pot fi schimbate pot fi de asemenea accesate

$$\text{var}(P) \subseteq \text{acc}(P) \subseteq \alpha P$$

Similar, definim $\text{acc}(e)$ mulțimea variabilelor ce apar în e . Astfel dacă P și Q sunt combinate prin \parallel trebuie ca

$$\text{var}(P) \cap \text{acc}(Q) = \text{acc}(P) \cap \text{var}(Q) = \{\}$$

Odată această condiție îndeplinită, nu mai contează dacă o asignare are loc înainte de combinarea paralelă sau în interiorul unui proces după ce acesta începe să activeze concurent.

$$\text{L14 } ((x:=e; P) \parallel Q) = (x:=e; (P \parallel Q))$$

în condițiile când $x \subseteq \text{var}(P) - \text{acc}(Q)$ și $\text{acc}(e) \cap \text{var}(Q) = \{\}$

O consecință imediată a acesteia este

$$(x:=e; P) \parallel (y:=f; Q) = (x, y:=e, f; (P \parallel Q))$$

în condițiile când $x \subseteq \text{var}(P) - \text{acc}(Q) - \text{acc}(f)$
 și $y \subseteq \text{var}(Q) - \text{acc}(P) - \text{acc}(e)$.

Cele de mai sus ne arată că restricțiile de alfabet protejează asignările dintr-un proces component al perechii concurente să nu poată interfera cu asignările din celălalt. Într-o implementare, secvențe de asignări pot fi executate atât împreună cât și întrețesut fără a le diferenția de acțiunile externe observabile ale procesului.

În fine, compunerea concurentă se distribuie cu operatorul condiție

$$\text{L15 } P \parallel (Q \nabla b \nabla R) = (P \parallel Q) \nabla b \nabla (P \parallel R)$$

când $acc(b) \cap var(P) = \{\}$.

Această lege afirmă încă o dată că nu contează dacă b este evaluat înainte sau după compunerea paralelă.

Vom trata acum problema care se naște când expresiile sunt nedefinite pentru anumite valori ale variabilelor ce le conțin. Dacă e este o listă de expresii, definim $\mathcal{D}e$ ca o expresie booleană care este adevărată când toți operandii lui e sunt în domeniile operatorilor lor. De exemplu, pentru numere naturale

$$\mathcal{D}(x+y) = (y > 0)$$

$$\mathcal{D}(y+1, z+y) = true$$

$$\mathcal{D}(e+f) = \mathcal{D}e \wedge \mathcal{D}f$$

$$\mathcal{D}(r-y) = y \leq r$$

Este rezonabil de insistat ca $\mathcal{D}e$ să fie totdeauna definit

$$\mathcal{D}(\mathcal{D}e) = true$$

În mod deliberat am lăsat complet nespecificat rezultatul unei încercări de a evalua expresii nedefinite – unde se poate întâmpla de fapt orice. Aceasta este exemplificat de utilizarea lui *CHAOS* în următoarele legi

$$L16 \quad (x:=e) = (x:=e \nmid \mathcal{D}e \nmid CHAOS)$$

$$L17 \quad P \nmid b \nmid Q = ((P \nmid b \nmid Q) \nmid \mathcal{D}b \nmid CHAOS)$$

Mai mult, legile L2, L5 și L12 necesită ușoare modificări.

$$L2' \quad (x:=e; x:=f(x)) = (x:=f(e) \nmid \mathcal{D}e \nmid CHAOS)$$

$$L5' \quad (P \nmid b \nmid P) = (P \nmid \mathcal{D}b \nmid CHAOS)$$

5.5.2 Specificații

O specificare a unui proces secvențial descrie nu numai urmele evenimentelor care au loc dar și relația între aceste urme, valorile inițiale ale variabilelor programului și valorile lor finale. Pentru indicarea valorii inițiale a unei variabile de program x vom utiliza chiar numele ei x . Pentru a indica valoarea finală adăugăm la x semnul \checkmark , ca de exemplu x^{\checkmark} . Valoarea lui x^{\checkmark} nu este observabilă până ce procesul nu se termină sau ultimul eveniment al urmei este \checkmark

Acest lucru face să nu specificăm nimic despre x^{\checkmark} , ci numai $\overline{ur_0} = \checkmark$.

Exemple

X1 Un proces care nu face nici o acțiune, dar adună 1 valorii lui x și se termină cu succes cu valoarea lui y neschimbată

$$ur = \Diamond \vee (ur = \langle \sqrt{} \rangle \wedge x^{\sqrt{}} = x + 1 \wedge y^{\sqrt{}} = y) \quad \square$$

X2 Un proces care se angajează într-un eveniment al cărui simbol este valoarea inițială a unei variabile x și apoi se termină cu succes lăsând valorile finale ale lui x și y egale cu cele inițiale.

$$ur = \Diamond \vee ur = \langle x \rangle \vee (ur = \langle x, \sqrt{} \rangle \wedge x^{\sqrt{}} = x \wedge y^{\sqrt{}} = y) \quad \square$$

X3 Un proces care memorează identitatea primului său eveniment ca valoarea finală a lui x

$$\#ur \leq 2 \wedge (\#ur = 2 \Rightarrow (ur = \langle x^{\sqrt{}}, \sqrt{} \rangle \wedge y^{\sqrt{}} = y)) \quad \square$$

Funcționarea corectă a unui proces depinde adesea de condițiile $S(x)$ ale valorilor inițiale pentru variabilele de program x . Acestea pot fi exprimate scriind $S(x)$ ca o ante-specificație.

X4 Un proces care divide un număr nenegativ x la unul pozitiv y și asignează câtul lui q și restul lui r

$$DIV = (y > 0 \Rightarrow ur = \Diamond \vee (ur = \langle \sqrt{} \rangle \wedge q^{\sqrt{}} = (x \div y) \wedge r^{\sqrt{}} = x - (q^{\sqrt{}} \times y) \wedge y^{\sqrt{}} = y \wedge x^{\sqrt{}} = x))$$

Fără condiție această specificație ar fi fost în general imposibilă. □

X5 Iată câteva specificații mai complexe care vor fi folosite mai târziu.

$$BUCLĂDIV = (ur = \Diamond \vee (ur = \langle \sqrt{} \rangle \wedge r = (q^{\sqrt{}} - q) \times y + r^{\sqrt{}} \wedge r^{\sqrt{}} < y \wedge x^{\sqrt{}} = x \wedge y^{\sqrt{}} = y)) \\ T(n) = r < n \times y \quad \square$$

Toate variabilele din aceste specificații și următoarele se referă la numere naturale astfel că scăderea nu e definită dacă al doilea operand este mai mare ca primul.

Vom formula acum legile care pun în evidență că un proces își satisface specificațiile. Fie $S(x, ur, x^{\sqrt{}})$ o specificație. Pentru a dovedi că *SKIP* satisface această specificație în mod clar, specificația trebuie să fie adevărată când urma este vidă. Mai mult, ea trebuie să fie adevărată când urma este $\langle \sqrt{} \rangle$ și valorile

finale ale tuturor variabilelor x^\vee sunt egale cu valorile lor inițiale. Aceste două condiții sunt de asemenea suficiente așa cum se afirmă în următoarea lege

L1 Dacă $S(x, \diamond, x^\vee)$
și $S(x, \langle \vee \rangle, x)$
atunci $SKIP \text{ sat } S(x, ur, x^\vee)$

X6 Cea mai puternică specificație satisfăcută de $SKIP$ este

$$SKIP_A \text{ sat } (ur = \diamond \vee (ur = \langle \vee \rangle \wedge x^\vee = x))$$

unde x este o listă a tuturor variabilelor din A iar x^\vee este o listă a variabilelor cu \vee . $X6$ este o consecință imediată a lui L1 și viceversa. \square

X7 $SKIP \text{ sat } (r \triangleleft y \Rightarrow (T(n+1) \Rightarrow BUCLĂDIV))$

Demonstrație

(1) Înlocuind ur cu \diamond în specificație avem

$$r \triangleleft y \wedge T(n+1) \Rightarrow \diamond = \diamond \vee \dots$$

care este o tautologie.

(2) Înlocuind ur cu $\langle \vee \rangle$ și valorile finale cu cele inițiale avem

$$r \triangleleft y \wedge T(n+1) \Rightarrow (\langle \vee \rangle = \diamond \vee (\langle \vee \rangle = \langle \vee \rangle \wedge x = x \wedge y = y \wedge \neg((q-q)xy+r) \wedge r \triangleleft y))$$

care este o teoremă banală. Acest rezultat va fi folosit în X10. \square

O precondiție a asignării cu succes $x := e$ este ca expresia e să fie definită. În acest caz, dacă P satisface o specificație $S(x)$, $(x := e; P)$ satisface aceeași specificație după modificarea care reflectă că valoarea inițială a lui x este e .

L2 Dacă $P \text{ sat } S(x)$
atunci $(x := e; P) \text{ sat } (\mathcal{D}e \Rightarrow S(e))$

Legea pentru asignarea simplă poate fi derivată din L2 înlocuind P cu $SKIP$ și folosind X6 și 5.2 L1.

L2A $x_0 := e \text{ sat } (\mathcal{D}e \wedge ur \neq \diamond \Rightarrow ur = \langle \vee \rangle \wedge x_0^\vee = e \wedge x_1^\vee = x_1 \wedge \dots)$

O consecință a lui L2A este aceea că pentru orice P , cea mai tare afirmație care se poate demonstra despre $(x := 1/0; P)$ este

$(x:1/0;P)$ sat true

Oricare ar fi dezideratul concret la care vrem să ajungem el nu poate fi atins pornind de la o asignare ilegală.

Exemple

X8 SKIP sat $(ur \neq \diamond \Rightarrow ur = \langle \sqrt{} \rangle \wedge q^{\sqrt{}} = q \wedge r^{\sqrt{}} = r \wedge y^{\sqrt{}} = y \wedge x^{\sqrt{}} = x)$

de aceea $(r := x - q \times y; SKIP)$ sat $(x \geq q \times y \wedge ur \neq \diamond \Rightarrow$

$$ur = \langle \sqrt{} \rangle \wedge q^{\sqrt{}} = q \wedge r^{\sqrt{}} = (x - q \times y) \wedge y^{\sqrt{}} = y \wedge x^{\sqrt{}} = x)$$

de aceea $(q := x + y; r := x - q \times y)$ sat $(y > 0 \wedge x \geq (x + y) \times y \wedge ur \neq \diamond \Rightarrow$

$$ur = \langle \sqrt{} \rangle \wedge q^{\sqrt{}} = (x + y) \wedge r^{\sqrt{}} = (x - (x + y) \times y) \wedge y^{\sqrt{}} = y \wedge x^{\sqrt{}} = x)$$

Specificarea din ultima linie este echivalentă lui DII' care a fost definit în X4. \square

X9 Presupunem că Λ sat $(T(n) \Rightarrow BUCL\check{A}DII)$

de aceea $(r := r - y; \Lambda)$ sat $(y \leq r \Rightarrow (r - y < n \times y \Rightarrow (ur = \diamond \vee ur = \langle \sqrt{} \rangle \wedge (r - y) = \dots)))$

deci $(q := q + 1; r := r - y; \Lambda)$ sat $(y \leq r \Rightarrow (r < (n + 1) \times y \Rightarrow BUCL\check{A}DII'))$

unde $BUCL\check{A}DII' = (ur = \diamond \vee ur = \langle \sqrt{} \rangle \wedge (r - y) = (q^{\sqrt{}} - (q + 1)) \times y + r^{\sqrt{}} \wedge r^{\sqrt{}} < y \wedge x^{\sqrt{}} = x \wedge y^{\sqrt{}} = y))$

Din algebra elementară a numerelor naturale

$$y \leq r \Rightarrow (BUCL\check{A}DII' \equiv BUCL\check{A}DII)$$

de aceea $(q := q + 1; r := r - y; \Lambda)$ sat $(y \leq r \Rightarrow (T(n + 1) \Rightarrow BUCL\check{A}DII'))$

Acest rezultat va fi folosit în X10. \square

Pentru compunerea secvențială generală este necesară o lege mult mai complicată în care urmele componentelor sunt compuse secvențial și starea inițială a celei de-a doua componente este identică cu starea finală a primeia. Totuși, valorile variabilelor în această stare intermediară nu sunt observabile, fiind asigurată numai existența unor astfel de valori

L3 Dacă P sat $S(x, ur, x^{\sqrt{}})$

și Q sat $T(x, ur, x^{\sqrt{}})$

și P nu diverge

atunci $(P; Q)$ sat $(\exists y, s, t. ur = (s; t) \wedge S(x, s, y) \wedge T(y, t, x^{\sqrt{}}))$

În această lege, x este o listă de variabile din alfabetele lui P și Q , x^\vee este o listă a variantelor cu \vee și y o listă cu același număr de variabile noi.

Specificarea condiției este aceeași ca pentru prima componentă dacă condiția este adevărată și ca pentru a doua dacă este falsă

L4 Dacă $P \text{ sat } S$ și $Q \text{ sat } T$
atunci $(P \nabla b \nabla Q) \text{ sat } ((b \wedge S) \vee (\neg b \wedge T))$

O altă formă a acestei legi este câte o dată mai convenabilă

L4A Dacă $P \text{ sat } (b \Rightarrow S)$ și $Q \text{ sat } (\neg b \Rightarrow S)$
atunci $(P \nabla b \nabla Q) \text{ sat } S$

Exemplu

X10 Fie $COND = (q := q + 1; r := r - y; X) \nabla r \geq y \nabla SKIP$
și $X \text{ sat } (T(n) \Rightarrow BUCLĂDIV)$
atunci $COND \text{ sat } (T(n+1) \Rightarrow BUCLĂDIV)$

Cele două condiții suficiente pentru această concluzie s-au demonstrat în X7 și X9, iar rezultatul decurge din L4A. \square

Demonstrarea specificării unei bucle folosește definiția recursivă dată în 5.5 D1 și în legea pentru recursivitatea fără gardă (3.7.1 L8). Dacă R este specificația dorită a buclei trebuie să găsim o specificare $S(n)$ astfel încât $S(0)$ să fie totdeauna adevărată și de asemenea

$$(\forall n. S(n)) \Rightarrow R$$

O metodă generală pentru a construi $S(n)$ este de a găsi un predicat $T(n, x)$ care descrie condițiile din starea inițială x astfel încât bucla să se termine cu siguranță în mai puțin de n pași. Deci definim

$$S(n) = (T(n, x) \Rightarrow R)$$

În mod clar, nici o buclă nu poate avea 0 pași astfel că dacă $T(n, x)$ a fost corect definită, $T(0, x)$ este falsă și deci $S(0)$ va fi adevărată. Rezultatul demonstrației asupra specificării buclei va fi $\forall n. S(n)$ sau

$$\forall n. (T(n, x) \Rightarrow R)$$

Deoarece n este o variabilă astfel aleasă ca să nu apară în R , aceasta este echivalent cu

$$\exists n. (T(n, x) \Rightarrow R)$$

Nu există nici o specificare mai tare decât $\exists n. T(n, x)$ ca precondiție astfel încât bucla să se termine într-un număr finit de pași

În fine, trebuie să dovedim că corpul buclei satisface specificația. Deoarece ecuația recursivă pentru o buclă implică o condiție, această activitate se desparte în două. De aceea avem legea generală

L5 Dacă $\neg T(0, x)$ și $T(n, x) \Rightarrow \neg h$
 și $SKIP \text{ sat } (\neg h \Rightarrow (T(n, x) \Rightarrow R))$
 și $(\neg X \text{ sat } (T(n, x) \Rightarrow R)) \Rightarrow ((Q \wedge \neg X) \text{ sat } (h \Rightarrow (T(n-1, x) \Rightarrow R)))$
 atunci $(b * Q) \text{ sat } ((\exists n. T(n, x)) \Rightarrow R)$

Exemplu

X11 Dorim să dovedim că programul pentru metoda lungă de împărțire prin diferență repetată (5.5 X5) satisface specificația lui *DI1*. Activitatea se desparte natural în două. A doua parte, mai dificilă de arătat, este că bucla satisface specificația corespunzător formulată și anume

$$(r \geq y) * (q := q + 1; r := r - y) \text{ sat } (y > 0 \Rightarrow BI \text{ 'CL' } DI1)$$

Mai întâi e necesar să formulăm condiția astfel ca bucla să se termine în mai puțin de n iterații

$$T(n) = r < n \wedge y$$

Aici $T(0)$ este evident fals. Clauza

$$\exists n. T(n)$$

este echivalentă cu

$$y > 0$$

care este precondiția ca bucla să se termine. Restul pașilor pentru demonstrarea buclei au fost făcuți în X7 și X5. În continuare demonstrația este un simplu exercițiu. \square

Legile date în acest paragraf sunt destinate unui calcul al corectitudinii totale pentru programele pur secvențiale care nu conțin intrări sau ieșiri. Dacă Q este un astfel de program, atunci o demonstrație a faptului că

$$Q \text{ sat } (P(x) \wedge ur \neq \langle \rangle \Rightarrow ur = \langle v \rangle \wedge R(x, x^v)) \quad (1)$$

stabilește că dacă $P(x)$ este adevărată pentru valorile inițiale ale variabilelor când Q este activ atunci Q se va termina și $R(x, x^V)$ va descrie relația dintre valorile inițiale ale lui x și valorile finale x^V . Astfel $(P(x), R(x, x^V))$ este o pereche precondiție/postcondiție în sensul lui Cliff Jones. Dacă $R(x^V)$ nu specifică valoarea inițială x , (1) este echivalentă cu

$$P(x) \Rightarrow wp(Q, R(x))$$

unde wp este precondiția cea mai slabă a lui Dijkstra.

Astfel în capul special al programelor necomunicante, metodele de demonstrare sunt echivalente matematic cu acelea deja familiare, cu toate că menționarea explicită a lui " $ur = \Diamond$ " și " $ur = \langle \sqrt{} \rangle$ " le face notațional mai stângace. Acest lucru suplimentar este desigur necesar și de aceea acceptabil când metodele sunt extinse la procese secvențiale comunicante.

5.5.3 Implementare

Stările inițiale și finale ale proceselor secvențiale pot fi reprezentate ca o funcție care face să-i corespundă fiecărui nume de variabilă valoarea sa. Un proces secvențial este definit ca o funcție care face să-i corespundă stării sale inițiale comportarea sa ulterioară. Terminarea cu succes este reprezentată de atomul "SUCCEȘ. Un proces care este gata de a se termina va accepta acest simbol căruia îi va corespunde nu un alt proces ci starea finală a variabilelor sale. Astfel procesul SKIP are ca parametru o stare inițială, acceptă atomul "SUCCEȘ ca singura acțiune și livrează starea inițială ca stare finală

$$SKIP = \lambda s. \lambda y. \text{ if } y \neq \text{"SUCCEȘ"} \text{ then "BLIP" else } s$$

O asignare este similară cu SKIP, cu deosebirea că starea inițială este puțin diferită

$$\text{asign}(x, e) = \lambda s. \lambda y. \text{ if } y \neq \text{"SUCCEȘ"} \text{ then "BLIP"} \\ \text{else actualizează}(s, x, e)$$

unde $\text{actualizează}(s, x, e) = \lambda y. \text{ if } y = x \text{ then eval}(e, s) \text{ else } s(y)$
și $\text{eval}(e, s)$ este rezultatul evaluării expresiei e în starea s .

Dacă e este nedefinită în starea s nu contează ce se întâmplă. Aici, pentru simplitate, am implementat numai o asignare simplă. Asignarea multiplă este puțin mai complicată.

Pentru a implementa compunerea secvențială este necesar să testăm mai întâi dacă primul operand s-a terminat cu succes. Dacă da, starea sa finală este trecută la al doilea operand. Dacă nu, prima acțiune este aceea a primului operand

$secvență(P, Q) =$

$\lambda s. \text{ if } P(s) ("SUCCES") \neq "BLIP" \text{ then } Q(P(s) ("SUCCES"))$
 $\text{ else } \lambda v. \text{ if } P(s)(v) = "BLIP" \text{ then } "BLIP"$
 $\text{ else } secvență(P(s)(v), Q)$

Implementarea condiției se face printr-o condiție

$condiție(P, b, Q) = \lambda s. \text{ if } eval(b, s) \text{ then } P(s) \text{ else } Q(s)$

Implementarea buclei ($b * Q$) este lăsată ca exercițiu.

De notat că definiția lui *secvență* dată mai sus este mai complicată decât aceea dată în paragraful 5.3.3, din cauză că are o stare s ca prim argument și trebuie să transfere această stare ca prim argument operanzilor. Din nefericire, o complexitate asemănătoare trebuie introdusă în definițiile tuturor celorlalți operatori dați în capitolele anterioare. O soluție mai simplă ar fi să modelăm variabilele ca procese subordonate. Dar aceasta ar fi probabil un lucru mai complicat și mai puțin eficient decât utilizarea memorării în acces aleator convențională. Când considerațiile de eficiență sunt adăugate celor matematice, sunt motive întemeiate pentru introducerea variabilelor de program asignabile ca noi concepte decât definirea lor prin conceptele deja introduse.

6 Resurse partajate

6.1 Introducere

În paragraful 4.5 am introdus conceptul de proces subordonat notat $(m:R)$, a cărui singură funcțiune este îndeplinirea serviciilor față de un singur proces principal S . Pentru aceasta am folosit notația

$$(m:R/S)$$

Presupunem acum că S conține sau constă din două procese concurente $(P||Q)$ și ambele procese P și Q necesită servicii ale aceluiași proces subordonat $(m:R)$. Din nefericire, nu este posibil ca P și Q să comunice amândouă cu $(m:R)$ de-a lungul acelorași canale, din cauză că aceste canale ar trebui să fie atât în alfabetul lui P cât și al lui Q și atunci definiția operatorului $||$ ar impune pentru comunicațiile lui P și Q cu $(m:R)$ transmiterea numai a acelorași mesaje simultan – care (cum s-a explicat și în 4.5 X6) este departe de efectul dorit. Ceea ce dorim este o posibilitate de întretesere a comunicațiilor între P și $(m:R)$ cu acelea între Q și $(m:R)$. În acest mod $(m:R)$ se comportă ca o resursă partajată lui P și Q . Fiecare din procese o va putea folosi independent și interacțiunile lor cu ea vor fi întretesute.

Când identitatea tuturor proceselor care partajează o resursă este cunoscută dinainte este posibil de aranjat ca fiecare proces partajant să folosească mulțimi de canale diferite pentru a comunica cu resursa partajată. Această tehnică s-a utilizat în problema mesei celor cinci filozofi (paragraful 2.5). Fiecare furculiță era partajată de doi filozofi vecini și valetul era partajat între toți cinci. Alt exemplu a fost 4.5 X6, în care un buffer era partajat între două procese, unul dintre ele utilizând numai canalul din stânga, celălalt numai canalul din dreapta. O metodă generală de partajare este realizată prin etichetare multiplă (paragraful 2.6.4) care efectiv creează destule canale separate pentru comunicația cu fiecare proces partajant. Comunicațiile individuale de-a lungul acestor canale sunt arbitrar întretesute. Dar această metodă necesită ca numele tuturor proceselor partajante să fie cunoscut dinainte și de aceea nu este adecvată pentru un proces subordonat care se intenționează să deservească nevoile unui proces principal ce la rândul lui se compune dintr-un număr de

sub-procese concurente. Acest capitol introduce tehnici pentru partajarea resurselor între mai multe procese, chiar când numărul și identitatea lor nu sunt cunoscute dinainte. Se vor ilustra tehnicile prin exemple din proiectarea unui sistem de operare.

6.2 Partajarea prin întrețesere

Problema evidențiată în paragraful 6.1 provine din utilizarea operatorului \parallel pentru descrierea comportării concurente a proceselor; problemă ce poate fi adeseori evitată utilizând în locul operatorului \parallel forma întrețesută de concurență ($P|||Q$). Aici, P și Q au același alfabet și comunicațiilor lor cu procese externe (partajate) este arbitrar întrețesută. Desigur, se interzice directă comunicare între P și Q , dar o comunicare indirectă poate fi stabilită cu ajutorul unui proces subordonat partajat așa cum se arată în 4.5 X6 și X2 de mai jos.

Exemple

X1 Subrutina partajată

dub:DUBLU ($P|||Q$)

Aici, atât P cât și Q pot conține apeluri ale procesului subordonat

$(dub.st\acute{a}nga!v \rightarrow dub.dreapta?x \rightarrow SKIP)$

Chiar când aceste perechi de comunicații de la P și Q sunt arbitrar întrețesute, nu există pericolul ca unul din procese să obțină accidental un răspuns care ar fi trebuit să fie primit de celălalt. Pentru a ne asigura de aceasta, toate sub-procesele procesului principal trebuie să respecte o strictă alternanță a comunicațiilor pe canalul din stânga cu comunicațiile pe cel din dreapta corespunzătoare procesului subordonat partajat. Din acest motiv, pare rezonabil să introducem o notație specializată a cărei utilizare exclusivă va garanta aplicarea disciplinei cerute. Notația în cauză este o reminescentă a unui apel procedural dintr-un limbaj de nivel înalt, cu deosebirea că parametrii valoare transmiși sunt precedați de ! iar parametrii rezultat de ?, astfel că

$dub!x?y = (dub.st\acute{a}nga!x \rightarrow dub.dreapta?y \rightarrow SKIP)$ □

Efectul dorit de partajare prin întrețesere este ilustrat prin următoarele serii de transformări algebrice. Când două procese partajate încearcă simultan să folosească subrutina partajată, perechile corespunzătoare de comunicații

sunt luate în ordine arbitrară, dar componentele unei perechi de comunicații cu un proces nu sunt niciodată separate de comunicația cu celălalt. Pentru ușurință folosim următoarele abrevieri

$$\left. \begin{array}{l} d!v \text{ pentru } d.st\acute{a}nga!v \\ d?v \text{ pentru } d.dreapta?v \end{array} \right\} \text{ în procesul principal}$$

$$\left. \begin{array}{l} !v \text{ pentru } dreapta!v \\ ?v \text{ pentru } st\acute{a}nga?v \end{array} \right\} \text{ în procesul subordonat}$$

Fie $D = ?x \rightarrow !(x+x) \rightarrow D$

și $P = d!3 \rightarrow d?v \rightarrow P(v)$

și $Q = d!4 \rightarrow d?z \rightarrow Q(z)$

și $R = (d:D // (P || Q))$

ca în X1 de mai sus

Atunci $P || Q = (d!3 \rightarrow \neg((d?v \rightarrow P(v)) || Q))$

$\square d!4 \rightarrow \neg(P || (d?z \rightarrow Q(z)))$

din 3.6.1 L7

Fiecare din procesele partajante începe cu emisia către procesul partajat. Procesului partajat i se oferă șansa de a alege între cele două procese principale. Dar procesul partajat dorește să le accepte pe oricare astfel că după mascarea alegerii devine nedeterminist

$$\begin{aligned} (d:D // (P || Q)) &= ((d:(!3+3 \rightarrow D)) // ((d?v \rightarrow P(v)) || Q)) & \left\{ \begin{array}{l} 4.5.1.L1 \\ 3.5.1.L5 \end{array} \right. \\ &\quad \square ((d:(!4+4 \rightarrow D)) || (P || (d?z \rightarrow Q(z)))) \\ &= (d:D // (P(6) || Q)) \square (d:D // (P || Q(8))) & \left\{ \begin{array}{l} \text{etc.} \end{array} \right. \end{aligned}$$

Procesul partajat oferă rezultatul său indiferent căruia din procesele partajante este gata să-l accepte. Deoarece cel puțin unul din ele stă în recepție, acesta va fi procesul ce a asigurat argumentul care generează rezultatul. De aceea este așa de importantă stricta alternare a transmisiei cu recepția în apelarea unei subrutine partajate.

X2 Structură de date partajate

Într-un sistem de rezervare a билетelor de avion rezervările de locuri se fac de către mulți funcționari a căror acțiuni sunt întretesute. Fiecare rezervare adaugă un pasager listei de zbor și returnează un răspuns dacă pasagerul a fost acceptat sau nu. În acest exemplu foarte simplu, mulțimea implementată în 4.5.X8 va servi ca proces subordonat partajat cu numele de la indicativul zborului

$RO109:SET//(... (FUNC||FUNC||...)...)$

Fiecare *FUNC* înregistrează un pasager prin apelul

$RO109!nr_pas?x$

care corespunde la

$(RO109.st\acute{a}nga!nr_pas \rightarrow RO109.dreapta?x \rightarrow SKIP)$

□

În aceste două exemple, fiecare ocazie de a utiliza resursa partajată implică exact două comunicații, una pentru a transmite parametrii și alta pentru a recepționa rezultate. După fiecare pereche de comunicații, procesul subordonat trece din nou în starea în care este gata să deservească alt proces sau pe același din nou. Dar frecvent vrem să ne asigurăm că o întreagă serie de comunicații are loc între două procese fără pericolul de a interfera cu un al treilea. De exemplu, un dispozitiv scump de transmisie ar putea fi partajat de mai multe procese concurente. De fiecare dată, un număr de linii constituind un fișier trebuie transmise consecutiv fără pericolul întreteserii liniilor trimise de alt proces. Pentru aceasta, transmisia fiecărui fișier trebuie precedată de un eveniment *ocupă* care obține utilizarea exclusivă a resursei, iar la sfârșit, resursa trebuie disponibilizată din nou prin evenimentul *eliberează*.

X3 Imprimantă partajată

$LP=ocup\acute{a} \rightarrow \mu X.(st\acute{a}nga?s \rightarrow h!s \rightarrow X \mid elibereaz\acute{a} \rightarrow LP)$

Aici, *h* este canalul care face legătura între procesul *LP* și hardware-ul imprimantei. După ocuparea imprimantei, procesul copie linii succesive de la canalul său stâng către canalul hardware până când un semnal de eliberare îl readuce în starea sa inițială în care este disponibil pentru alte procese. Acest proces este utilizat ca o resursă partajată

$lp:LP// \dots (P||Q) \dots$

În interiorul lui *P* și *Q* transmisia unor linii dintr-un fișier este încadrată de evenimentele *lp.ocupă* și *lp.eliberează*

$lp.ocup\acute{a} \rightarrow \dots lp.st\acute{a}nga!"C.IONESCU" \rightarrow \dots$
 $lp.st\acute{a}nga!linia_urm\acute{a}toare \rightarrow \dots lp.elibereaz\acute{a} \rightarrow \dots$

□

X4 O îmbunătățire a lui X3

Când o imprimantă este partajată între mai mulți utilizatori, zona hârtiei conținând fiecare fișier listat trebuie detașată manual dintre eventualele fișiere anterioare și următoare la sfârșitul fiecărei tipărituri conform cu X3. De aceea, de obicei, hârtia de imprimantă este împărțită în pagini care sunt separate de perforații și hardware-ul imprimantei permite operația *avans* care mișcă hârtia rapid către sfârșitul paginii curente – sau mai bine zis la începutul unei pagini noi. Pentru a ajuta separarea, fișierele trebuie cadrate în pagină și trebuie imprimat un rând complet de asteriscuri la sfârșitul ultimei pagini a fișierului precum și la începutul primei pagini. Pentru a evita confuziile nu se permite imprimarea unei linii complete de "*" în mijlocul fișierului.

$$LP = (h!avans \rightarrow h!asteriscuri \rightarrow ocupă \rightarrow h!asteriscuri \rightarrow \\ \mu X. (stânga?s \rightarrow \text{if } s \neq asteriscuri \text{ then } (h!s \rightarrow X) \text{ else } X \\ | eliberează \rightarrow LP))$$

Această versiune de *LP* are aceeași funcționalitate ca cea anterioară. \square

În ultimele două exemple, utilizarea evenimentelor *ocupă* și *eliberează* evită întreteserea arbitrară de linii din fișiere distincte și aceasta fără pericolul de blocaj. Dar dacă mai mult de o resursă este partajată în această manieră, riscul blocajului nu poate fi ignorat.

X5 Blocaj

Ann și Mary sunt bune prietene și bucătărese. Ele folosesc în comun o cratiță și o tigaie pe care le solicită, utilizează și eliberează după cum au nevoie

$$VASE = (ocupă \rightarrow utilizează \rightarrow utilizează \rightarrow \dots \rightarrow eliberează \rightarrow VASE) \\ cratiță: VASE // tigaie: VASE \parallel ANN \parallel MARY)$$

Ann gătește potrivit unei rețete care cere întâi cratița și apoi tigaia, în timp ce Mary are nevoie întâi de tigaie și apoi de cratiță

$$ANN = \dots cratiță. ocupă \rightarrow \dots \rightarrow tigaie. ocupă \rightarrow \dots \\ MARY = \dots tigaie. ocupă \rightarrow \dots \rightarrow cratiță. ocupă \rightarrow \dots$$

Din nefericire, gospodinele decid să înceapă aproximativ în același timp. Fiecare solicită primul vas corespunzător, dar când au nevoie de al doilea își dau seama că nu îl pot avea deoarece este utilizat de prietenă.

Povestea lui Ann și Mary poate fi ilustrată grafic (fig. 6.1) unde timpul lui Ann este pe axa verticală iar a lui Mary pe cea orizontală. Sistemul pornește din colțul din stânga jos, originea timpului pentru ambele gospodine. De fiecare dată când Ann face o acțiune, sistemul se mișcă în sus un pas. De fiecare dată când Mary face o acțiune, sistemul se mișcă la dreapta un pas.

Traectoria urmată arată clar întrepesarea acțiunilor lui Ann și Mary. Din fericire, această traectorie ajunge în colțul din dreapta sus unde ambele gospodine se bucură că masa este gata.

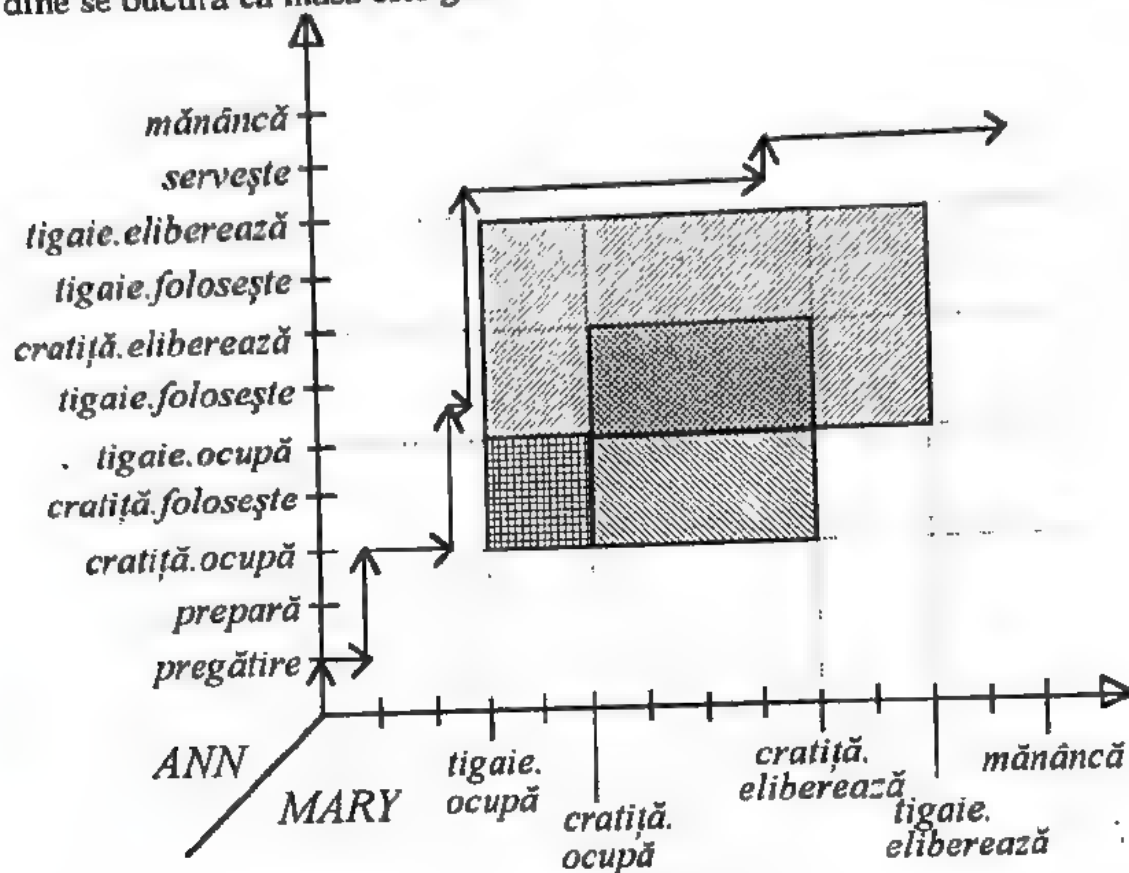





Figura 6.1

Dar acest sfârșit fericit este nesigur. Deoarece nu pot folosi simultan un vas partajat există anumite regiuni în spațiul stărilor prin care traectoria nu poate trece. De exemplu, în regiunea hașurată cu  ambele gospodine ar folosi tigaia și aceasta nu este posibil. Analog, excluderea în utilizarea simultană a cratiței interzice intrarea în regiunea hașurată cu . Astfel dacă traectoria atinge marginea uneia din zonele interzise poate numai să urmeze marginea (vertical sau orizontal). În acest timp, una din gospodine așteaptă eliberarea vasului de către cealaltă.

Să considerăm acum zona cu . Dacă traectoria intră vreodată în această zonă va sfârși inevitabil în blocaj în colțul din dreapta sus al zonei. Scopul desenului este de a arăta că pericolul blocajului provine numai din zona interzisă din întreaga regiune mărginită : celelalte zone sunt sigure. Singura modalitate de a preveni blocajul este de a extinde regiunea interzisă și în zonele periculoase. O tehnică ar fi să introducem o resursă artificială în plus care ar fi solicitată înaintea oricărui vas și nu trebuie eliberată până ce ambele vase nu s-au eliberat. Această soluție este similară celei găsite prin introducerea valetului în povestea mesei filozofilor (paragraful 2.5.3.), unde permisiunea de a sta jos este un fel de resursă din care numai patru copii sunt par-

taiate de cei cinci filozofi. O soluție mai ușoară este de a insista ca gospodina care vrea ambele vase să solicite mai întâi tigaia. Acest exemplu este datorat lui E. W. Dijkstra. \square

Soluția cea mai ușoară sugerată pentru exemplul anterior se poate generaliza la orice număr de utilizatori și resurse. În condițiile în care este o ordine în care utilizatorii solicită resursele nu există riscul de blocaj. Utilizatorii ar trebui să-și elibereze resursele cât mai repede posibil după ce le-au folosit, ordinea de eliberare necontând. Utilizatorii pot solicita resurse și într-o ordine oarecare, în condițiile în care în momentul solicitării toate resursele sunt deja eliberate fiind ordonate apoi corespunzător. Respectarea acestei discipline a solicitării și eliberării resurselor poate fi verificată adesea printr-o vizualizare a textului proceselor utilizator.

6.3 Partajarea memoriei

Obiectivul acestui paragraf este de a aduce argumente împotriva utilizării memoriei partajate. Paragraful poate fi sărit de cei care sunt deja convinși de acest lucru.

Comportarea sistemelor de procese concurente poate fi ușor implementată pe un singur computer convențional printr-o tehnică numită *partajarea timpului* (*timesharing*), în care un singur procesor execută pe rând fiecare din procese, cu schimbarea procesului prin intermediul unei întreruperi de la un dispozitiv extern sau de la un ceas sistem. În această implementare este foarte ușor să permitem proceselor concurente să partajeze locațiile de memorie comună care sunt accesate și asignate simplu cu ajutorul instrucțiunilor mașină uzuale din programul corespunzător fiecărui proces.

O locație de memorie partajată poate fi modelată în teoria noastră ca o variabilă partajată (4.2 X7) cu un nume simbolic adecvat, de exemplu

$(\text{contor}.\text{VAR}(\text{contor}.\text{stânga}|0 \rightarrow (P|||Q)))$

Memorarea partajată poate fi ușor deosebită de o memorare locală descrisă în 5.5. Simplitatea legilor pentru formalismul proceselor secvențiale derivă numai din faptul că variabila este actualizată de cel mult un proces și aceste legi nu tratează multiplele pericole care provin din întreteserea arbitrară a asignărilor în diferite procese.

Aceste pericole sunt mai clar ilustrate de următorul exemplu.

X1 Interferența

Variabila partajată *contor* este utilizată pentru a ține socoteala numărului total de apariții ale unui eveniment important. La fiecare apariție a evenimentului,

procesul relevant P sau Q încearcă să actualizeze *contor* prin perechea de comunicații

$\text{contor.dreapta?}x; \text{contor.stânga!}(x+1)$

Din nefericire aceste două evenimente de comunicații pot fi întrepesute cu o pereche identică de evenimente de comunicații de la un alt proces, rezultând secvența

$\text{contor.dreapta?}x \rightarrow \text{contor.dreapta?}y \rightarrow \text{contor.stânga!}(y+1) \rightarrow$
 $\text{contor.stânga!}(x+1) \rightarrow \dots$

Ca o consecință, valoarea lui *contor* este incrementată numai de unul din procese, în loc de două. Acest fel de eroare este cunoscut ca interferență și este o eroare frecventă în proiectarea proceselor care partajează memorie comună. Mai mult, apariția erorii este complet nedeterministă. Ea nu este reproductibilă sigur și de aceea este imposibil (aproape) să diagnosticăm eroarea prin tehnici de testare convenționale. Ca un principal rezultat, eu suspectez că există câteva sisteme de operare de largă utilizare care regulat produc ușoare inexactități la statistici și contabilizări. \square

O posibilă soluție a acestei probleme este să ne asigurăm că nici o schimbare a procesului nu are loc în timpul unei secvențe de acțiuni, ceea ce înseamnă o protecție la întrepesere. O astfel de secvență este cunoscută ca *regiune critică*. La implementarea pe un singur procesor, excluderea necesară este adesea realizată prin inhibarea tuturor întreruperilor pe perioada regiunii critice. Soluția are efectul nedorit al întârzierii răspunsului la întreruperi și, mai rău, se prăbușește când este adăugată o a doua unitate de procesare în computer.

O soluție mai bună a fost sugerată de E.W. Dijkstra în introducerea sa la semafoarele de excludere binare. Un semafor poate fi descris ca un proces care se angajează succesiv în acțiunile P și V

$SEM = (P \rightarrow V \rightarrow SEM)$

Aceasta este declarată ca o resursă partajată

$(\text{mutex}; SEM // \dots)$

Fiecare proces, la intrarea în regiunea critică, trebuie să trimită semnalul

$\text{mutex}.P$

și la ieșirea din regiunea critică trebuie să se angajeze în evenimentul

mutex.V

Astfel regiunea critică în care *contor* este incrementată ar apărea

```
mutex.P →
contor.dreapta?x → contor.stânga!(x+1) →
mutex.V → ...
```

În condițiile în care toate procesele respectă această disciplină este imposibil ca două procese să interfere între ele actualizând *contor*. Dar dacă unul din procese omite *P* sau *V* sau le ia în ordine greșită efectul va fi haotic și se va risca o eroare subtilă, dezastruoasă (sau mai rău).

O cale mai sigură de a evita interferența este de a clădi protecția necesară prin însăși modelarea memoriei partajate, profitând de faptul că știm cum o vom utiliza. De exemplu, dacă o variabilă trebuie folosită numai pentru socoteli, atunci operația care o incrementează ar putea fi o operație atomică

contor.sus

și resursa partajată ar putea fi proiectată ca MM_0 (1.1.4 X2)

contor.MM₀(...P||Q...)

De fapt sunt destule motive pentru a recomanda ca fiecare resursă partajată să fie proiectată special pentru scopul ei iar în proiectarea unui sistem folosind concurența memoria principală nu trebuie partajată. Aceasta nu numai că evită pericolul interferenței accidentale, ci de asemenea, permite realizarea unui proiect care poate fi implementat eficient pe rețele cu elemente de procesare distribuite ca de altfel și pe un singur procesor sau multiprocesor cu memorie partajată fizic.

6.4 Resurse multiple

În paragraful 6.2 am descris cum un număr de procese concurente cu comportare diferită ar putea partaja un singur proces subordonat. Fiecare proces partajant respectă o disciplină a alternării transmisiei cu recepția sau alternării semnalelor de ocupare și eliberare, pentru a se asigura că în orice moment resursa este utilizată de cel mult unul din potențialele procese partajante. Astfel de resurse sunt cunoscute ca *reutilizabile serial*. În acest paragraf introducem masive de procese pentru a reprezenta o resursă multiplă cu comportare identică. Un indice în masiv ne asigură că fiecare element comunică în siguranță cu procesul care l-a solicitat.

De aceea vom utiliza pe scară largă indici și operatori indexați cu semnificație evidentă. De exemplu

$$\begin{aligned} \|P_i &= (P_0 \| P_1 \| \dots \| P_{11}) \\ i < 12 \\ \| \| P &= (P \| \| P \| \| P \| \| P) \\ i < 4 \\ \| P_i &= (P_0 \| P_1 \| \dots) \\ i \geq 0 \\ \Box (f(i) \rightarrow P_i) &= (f(0) \rightarrow P_0 | f(1) \rightarrow P_1 | \dots) \\ i \geq 0 \end{aligned}$$

În ultimul exemplu se impune ca f să fie injectivă, astfel încât alegerea între alternative să fie făcută numai de mediu.

Exemple

X1 Subrutină reentrantă

O subrutină partajată care este folosită serial poate fi utilizată numai de un singur proces apelant odată. Dacă execuția subrutinei necesită calcule considerabile, ar putea fi întârzieri semnificative în procesele apelante. Dacă sunt disponibile mai multe procesoare, există suficiente motive de a permite unor copii ale subrutinei să se lanseze concurrent pe diferite procesoare. O subrutină capabilă de mai multe copii concurente este cunoscută ca *reentrantă* și este definită ca un masiv de procese concurente

$$dub: (\| (i: DUBLU) \forall \dots \\ i < 27$$

Un apel tipic al acestei subrutine ar putea fi

$$dub.3.st\acute{a}nga!30 \rightarrow dub.3.dreapta?y \rightarrow SKIP)$$

Utilizarea indexului 3 ne asigură că rezultatul apelului este obținut de la aceeași instanță a lui *dub* căreia i-au fost trimise argumentele, chiar dacă alt proces concurrent ar putea apela în același timp altă instanță din masiv, rezultând o întrepesere a mesajelor

$$dub.3.st\acute{a}nga.30, \dots dub.2.st\acute{a}nga.20, \dots dub.3.dreapta.60, \dots dub.2.dreapta.40, \dots$$

Când un proces apelează o subrutină reentrantă nu contează care element din masiv va răspunde la apel, oricare va fi liber va fi la fel de bun. Din acest

motiv, decât să specificăm un index particular, de exemplu 2 sau 3, un proces particular ar putea lăsa ca selecția să se facă arbitrar prin construcția

$$\square(dub.i.st\acute{a}nga!30 \rightarrow dub.l.dreapta?y \rightarrow SKIP)$$

$i \geq 0$

Se observă respectarea disciplinei esențiale ca același index să fie folosit pentru trimiterea argumentelor și (imediat după) pentru recepționarea rezultatului. \square

În exemplul arătat mai sus există o limită arbitrară de 27 de activări simultane ale subrutinei. Deoarece este relativ echitabil de aranjat ca un singur procesor să-și împartă atenția între un număr mare de procese, astfel de limite arbitrare pot fi evitate prin introducerea unui masiv infinit de procese concurente

$$dub:(||i:D)$$

$i \geq 0$

unde D poate fi proiectat să servească numai un singur apel și apoi să se oprească

$$D = st\acute{a}nga?x \rightarrow dreapta!(x+x) \rightarrow STOP$$

O subrutină fără limite în reentranta ei este cunoscută ca *procedură*.

Intenția utilizării unei proceduri este ca efectul fiecărui apel

$$\square(dub.i.st\acute{a}nga!30 \rightarrow dub.l.dreapta?y \rightarrow SKIP)$$

$i \geq 0$

să fie identic cu apelul unui proces subordonat D declarat adiacent imediat apelului

$$(dub:D(dub.st\acute{a}nga!30 \rightarrow dub.dreapta?y \rightarrow SKIP))$$

Această tehnică este cunoscută ca un apel *local* de procedură, deoarece sugerează execuția procedurii pe același procesor ca și procesul apelant, în timp ce apelul unei proceduri partajate este cunoscut ca apel *la distanță*, deoarece sugerează execuția pe un posibil procesor aflat la distanță. Deoarece efectul unor apeluri locale sau la distanță se dorește a fi același, motivele pentru utilizarea apelului la distanță pot fi numai politice sau economice – de exemplu ținerea codului procedurii secret sau lansarea ei pe o mașină cu fa-

cilități speciale care sunt prea scumpe de asigurat pe mașina pe care se rulează procesele utilizator.

Un exemplu tipic de facilitare scumpă este un suport de înmagazinare de mare volum, ca de exemplu un disc sau memoria cu bule.

X2 Memorie partajată secundară

Un mediu de memorare este împărțit în S sectoare care pot fi citite și scrise independent. Fiecare sector poate reține o cantitate de informație pe care-o recepționează prin canalul din stânga și o emite prin canalul din dreapta. Din nefericire, mediul de memorare este implementat într-o tehnologie cu citit distructiv, astfel că fiecare bloc scris poate fi citit o singură dată. Astfel, fiecare sector se comportă mai degrabă ca *COPIE* (4.2 X1) decât ca *VAR* (4.2 X7). Întregul mediu de memorare secundară este un masiv de astfel de sectoare, indexat după numere mai mici ca S

$$SMEM = \parallel_{i < S} i: COPIE$$

Această memorie se dorește a fi utilizată ca proces subordonat

$$(mems: SMEM // \dots)$$

În procesul principal, memoria poate fi folosită prin comunicații

$$mems.i.st\acute{a}nga!sec \rightarrow \dots mems.i.dreapta?y \rightarrow \dots$$

Memoria secundară poate fi de asemenea partajată de procese concurente. În acest caz acțiunea

$$\square_{i < S} (mems.i.st\acute{a}nga!sec \rightarrow \dots)$$

va solicita simultan un sector liber arbitrar cu numărul i și va scrie valoarea sec în el. Similar, $mems.i.dreapta?x$ va face într-o singură acțiune citirea conținutului sectorului i în x și eliberarea acestui sector pentru altă folosire cu altă ocazie, foarte posibil de un alt proces. Această simplificare este motivul real pentru utilizarea lui *COPIE* ca model de comportare pentru fiecare sector. Povestea cu citirea distructivă este doar o poveste. \square

Desigur, partajarea cu succes a memoriei secundare necesită o disciplină extremă din partea proceselor partajante. Un proces poate recepționa de la un sector numai dacă același proces a emis de curând către același sector și fiecare emisie trebuie eventual urmată de o recepție. Nerespectarea unei astfel de discipline va conduce la blocaj sau mai rău la confuzie. Metode de întărire a

acestei discipline fără complicații vor fi introduse după următorul exemplu și vor fi complet ilustrate în proiectarea ulterioară de module ale unui sistem de operare (paragraful 6.5)

X3 Două imprimante

Două imprimante identice sunt disponibile pentru a servi cererile unei colecții de procese utilizator. Ambele necesită protecție la întretesere care a fost asigurată de LP (6.2 X4). De aceea declarăm un masiv de două instanțe a lui LP fiecare indexat de un număr natural indicând poziția în masiv.

$$LP2 = (0:LP || 1:LP)$$

Acestui masiv îi poate fi dat un nume pentru utilizarea ca resursă partajată

$$(lp:LP2 || \dots)$$

Fiecare instanță a lui LP este acum prefixată de două ori, odată prin nume, odată prin indice. Astfel, comunicațiile cu procesul utilizator au trei sau patru componente, de exemplu

$$lp.0.ocupă, lp.1.stânga."C.IONESCU", \dots$$

Ca în cazul unei proceduri reentrante, când un proces are nevoie să solicite una din resursele identice dintr-un masiv, într-adevăr nu contează care element al masivului va fi selectat cu o anumită ocazie. Orice element care este gata să răspundă la semnalul de solicitare va fi acceptabil. O formă generală de alegere va face selecția arbitrară cerută

$$\square (lp.i.ocupă \rightarrow \dots lp.i.stânga!x \rightarrow \dots lp.i.eliberează \rightarrow SKIP)$$

20

Aici, acțiunea inițială $lp.i.ocupă$ va solicita oricare dintre cele două procese LP care este gata pentru acest eveniment. Dacă nici unul nu-i gata, procesul solicitant va aștepta. Dacă amândouă sunt gata, alegerea între ele este nedeterministă. După solicitarea inițială variabila mărginită i ia ca valoare indexul resursei selectate și toate comunicațiile ulterioare vor fi corect direcționate către această resursă. \square

Când o resursă partajată a fost solicitată pentru o folosire temporară într-un proces, resursa se dorește a se comporta exact ca un proces subordonat declarat local, comunicând numai cu procesul său utilizator. De aceea vom adapta și rescrie familiara notație pentru subordonare

$$(fispropriu::lp/\dots fispropriu.stânga!x\dots)$$

în loc de mai complexa construcție

$$\square (lp.i.ocupă \rightarrow \dots lp.i.stânga!x \rightarrow \dots; lp.i.eliberează \rightarrow SKIP) \\ i \geq 0$$

Aici, numele local *fispropriu* a fost introdus pentru a desemna numele indexat *lp.i* iar ocuparea și eliberarea au fost convenabil suprimate. Noua notație "::" se numește *subordonarea la distanță*. Se observă diferența față de deosebit de familiara notație ":" prin aceea că în partea dreapta nu apare un proces complet ci numele unui masiv de procese ce se pot afla la distanță.

X4 Două fișiere de ieșire

Un proces utilizator necesită utilizarea simultană a două imprimante prin transmiterea către ele a două fișiere de ieșire *f1* și *f2*

$$(f1::lp/(f2::lp/...f1.stânga!s1 \rightarrow f2.stânga!s2 \rightarrow \dots))$$

Aici, procesul utilizator întrețese transmiterea liniilor către cele două fișiere diferite dar fiecare linie este tipărită pe imprimanta corespunzătoare. Desigur, blocajul va apare sigur la orice încercare de a declara *trei* imprimante (fizic tot două) simultan. De asemenea un rezultat asemănător apare în cazul când se declară două imprimante simultan în fiecare din cele două procese concurente, cum se vede bine din istoria lui Ann și Mary (6.2 X5). \square

X5 Fișier de lucru (manevră)

Un fișier de lucru este folosit la citire pentru transmiterea unei secvențe de blocuri. Când transmisia este terminată, fișierul este rebobinat și întreaga secvență de blocuri este citită din nou de la început. Când toate blocurile au fost citite, fișierul manevră va genera numai semnalul *gol*, nefiind posibile citirile sau scrierile ulterioare. Astfel, un fișier de manevră se comportă ca un fișier de lucru pe bandă magnetică care trebuie rebobinat înainte de a fi citit. Semnalul *gol* servește ca marker de end-of-file

MANEV=SCRIE

SCRIE_s=(stânga!x→SCRIE_s^<∞>|rebobinare→CITEȘTE_s)

CITEȘTE_{<∞>_s=(dreapta!x→CITEȘTE_s)}

CITEȘTE_◇=(gol→CITEȘTE_◇)

Acest proces poate fi utilizat convenabil ca un proces subordonat nepartajat

fispropriu:MANEV//... fispropriu.stânga!v ... fispropriu.rebobinare ...
... (fispropriu.dreapta!x→...
|fispropriu.gol→...) ...)

Ne vom referi mai târziu la cele expuse mai sus ca la un model de proces partajat. \square

X6 Fișiere de manevră în memoria secundară

Fișierul de manevră descris în X5 poate fi ușor implementat prin reținerea secvenței de blocuri în memoria principală (RAM) a computerului. Dar dacă blocurile sunt mari și secvența lungă, aceasta ar fi o utilizare neeconomică a memoriei principale și ar fi mai bine să memorăm blocurile într-o memorie secundară. Deoarece orice bloc într-un fișier de manevră este citit și scris o singură dată, o memorie secundară (X2) cu citit destructiv va fi cea mai potrivită. Un fișier de lucru ordinar (ținut în memoria principală) este utilizat pentru a reține secvența de indici ale sectoarelor memoriei secundare unde se găsesc blocurile reale cu informație. Avem astfel certitudinea că sunt recitite corect blocurile, în secvența corespunzătoare

$$SMANEV = (tabpag.MANEV //$$

$$\mu X.(stanga?x \rightarrow (\bigwedge_{i < S} mems.i.stanga!x \rightarrow tabpag.stanga!i \rightarrow X)$$

$$| rebobinare \rightarrow tabpag.rebobinare \rightarrow$$

$$\mu Y.(tabpag.dreapta?i \rightarrow mems.i.dreapta?x \rightarrow dreapta!x \rightarrow Y$$

$$| tabpag.gol \rightarrow gol \rightarrow Y)))$$

SMANEV folosește numele *mems* pentru a adresa o memorie secundară (X2) ca un proces subordonat. Putem descrie aceasta prin

$$MANEVS = (mems.SMEM // SMANEV)$$

MANEVS poate fi utilizat ca un simplu proces subordonat nepartajat în exact același fel ca și fișierul de lucru din X5

$$(fispropriu.MANEVS // ... fispropriu.stanga!v ...)$$

Efectul este exact același ca și la utilizarea lui *MANEV* cu deosebirea că lungimea maximă a fișierului de lucru este limitată la *S* blocuri. \square

X7 Fișiere de manevră folosite serial

Să presupunem că vrem să partajăm fișierul de manevră din memoria secundară prin întreteserea între mai mulți utilizatori care îl vor solicita, utiliza și elibera câte unul o dată, în maniera unei imprimante partajate (6.2 X3). Pentru acest scop trebuie să adaptăm *SMANEV* pentru a accepta semnalele *ocupă* și *eliberează*. Dacă un utilizator eliberează fișierul înainte de a termina citirea există pericolul ca blocurile necitite din memoria secundară să nu mai fie accesate. Acest pericol poate fi evitat printr-o buclă care citește suplimentar aceste blocuri și le eliberează (citire în gol)

$$SCAN = \mu X. (tabpag.dreapta?i \rightarrow mems.i.dreapta?x \rightarrow X \\ | tabpag.gol \rightarrow SKIP)$$

Un fișier de manevră partajat întâi își captează utilizatorul și apoi se comportă ca *SMANEV*. Semnalul *eliberează* provoacă o întrerupere (paragraful 5.4) procesului *SCAN*

$$SMANPAR = ocupă \rightarrow (SMANEI \wedge (eliberează \rightarrow SCAN))$$

Fișierul de manevră reutilizabil serial este modelat de bucla simplă

**SMANPAR*

care folosește *SMEM* ca un proces subordonat

$$mems.SMEM // *SMANPAR \quad \square$$

X8 Fișiere de manevră multiplexat

În cele două exemple anterioare, la un moment dat este utilizat numai un singur fișier de manevră. O memorie secundară este de obicei suficient de mare pentru a permite mai multor fișiere de manevră să coexiste simultan, fiecare ocupând o mulțime disjunctă de sectoare disponibile. Memoria secundară poate de aceea să fie partajată între un număr suficient de mare (nemărginit) de fișiere de manevră. Fiecărui fișier de manevră *i* se alocă un sector când are nevoie prin scrierea unui bloc de informații în el și îl eliberează prin citirea informației din acel sector. Memoria secundară este partajată prin tehnica etichetării multiple (paragraful 2.6.4) utilizând ca etichete aceiași indici (numere naturale) care sunt utilizați în construcția unui masiv de procese partajante

$$SISFIS = N : (mems.SMEM) \parallel_{i \geq 0} (i : SMANPAR)$$

unde $N = \{i \mid i \geq 0\}$.

Acest sistem de fișiere se intenționează a se folosi ca un proces subordonat partajat, între orice număr de utilizatori întrețesuți

$$sisfis.SISFIS // \dots (UTIL1 \parallel \dots \parallel UTIL2 \parallel \dots)$$

Fiecare utilizator poate obține un fișier manevră, utiliza și elibera prin subordonare la distanță

*fispropriu::sisfis/(...fispropriu.stânga!v ... fispropriu.rebobinare ...
fispropriu.dreapta?x ...)*

intenționându-se (în afară de limitarea de resurse) obținerea aceluiași efect ca și prin subordonarea simplă a unui fișier manevră propriu (X5)

*(fispropriu:MANEV//...fispropriu.stânga!v ... fispropriu.rebobinare ...
fispropriu.dreapta?x ...)* □

Structura sistemului de fișiere (X8) și modul său de utilizare este un model de soluție generală a problemei partajării unui număr limitat de resurse reale (sectoare din memoria secundară) între un număr necunoscut de utilizatori. Utilizatorii nu comunică direct cu resursele ci există o *resursă virtuală* intermediară (*SMANPAR*) pe care ei (utilizatorii) o vor declara și utiliza ca și cum ar-fi un proces subordonat propriu. Funcția resursei virtuale este dublă :

- (1) asigură o interfață clară și simplă cu utilizatorul. În acest exemplu, *SMANPAR* reunește într-un singur fișier de manevră continuu o mulțime de sectoare dispersate din memoria secundară.
- (2) garantează un acces într-adevăr disciplinat la resursele reale. De exemplu, *SMANPAR* asigură că fiecare utilizator citește numai din sectoarele alocate lui și nu poate uita de eliberarea sectoarelor când se termină lucrul cu fișierul său de manevră.

Punctul (1) ne asigură că disciplina punctului (2) este mai puțin dureroasă.

Paradigma resurselor reale și virtuale este foarte importantă în proiectarea unui sistem cu partajarea resurselor. Definiția matematică a paradigmei este foarte complicată deoarece utilizează o mulțime nemărginită de numere naturale pentru implementarea creării dinamice necesară noilor procese virtuale și noilor canale prin care să se comunice cu ele. Într-o implementare practică pe calculator, acestea ar putea fi reprezentate prin blocuri de control, pointeri la înregistrările completate, etc. Pentru a utiliza paradigma efectiv, este cu siguranță mai bine să uităm metoda de implementare. Dar pentru acei care vor să o înțeleagă mai complet înainte de a o uita, următoarea explicație pentru X8 poate fi de ajutor.

Într-un proces utilizator un fișier manevră este creat printr-o subordonare la distanță

*fispropriu::sisfis/(...fispropriu.stânga!v ... fispropriu.rebobinare ...
fispropriu.dreapta?x ...)*

Din definiția subordonării la distanță aceasta este echivalentă cu

(\square sisfis.i.ocupă→
 $i \geq 0$ sisfis.i.stânga!v ... sisfis.i.rebobinare .. sisfis.i.dreapta!x
 sisfis.i.eliberează→SKIP)

Astfel toate comunicațiile între *sisfis* și utilizatorii săi încep cu *sisfis.i...*, unde *i* este indexul instanței particulare a lui *SMANPAR* care a fost obținută de un utilizator oarecare într-o ocazie oarecare. Mai mult, fiecare ocazie a utilizării sale este încadrată de o pereche corespunzătoare de semnale

(sisfis.i.ocupă→ ... sisfis.i.eliberează)

Din punctul de vedere al procesului subordonat, fiecare fișier de manevră virtual începe cu obținerea utilizatorului său și continuă potrivit schemei specificate în X6 și X7

(ocupă→stânga!x ... rebobinare ... dreapta!v ... eliberează)

Toate celelalte comunicații ale fișierului de manevră virtual sunt cu procesul subordonat *SMEM* și sunt mascate utilizatorului. Fiecare instanță a fișierului de manevră virtual este indexat cu un index diferit *i* și apoi numit cu numele *sisfis*. Astfel comportarea vizibilă externă a fiecărei instanțe este

(sisfis.i.ocupă→
 sisfis.i.stânga!x ... sisfis.i.rebobinare ... sisfis.i.dreapta!v ...
 sisfis.i.eliberează)

Aceasta se potrivește exact schemei de comunicare a utilizatorului descrisă în paragraful precedent. Perechile corespunzătoare de semnale *ocupă* și *eliberează* ne asigură că nici un utilizator nu poate interfera cu un fișier de manevră care a fost obținut de un alt utilizator.

Trecem acum la comunicațiile din *SISFIS* între masivul de fișiere de manevră virtuale și memoria secundară. Acestea sunt mascate utilizatorului și nici măcar nu au numele *sisfis* atașat lor. Evenimentele relevante sunt

i.mems.j.stânga.v înseamnă comunicarea unui bloc *v* de la elementul *i* din masivul de fișiere de manevră către sectorul *j* din memoria secundară.
i.mems.j.dreapta.v înseamnă comunicarea în direcția inversă.

Fiecare sector al memoriei secundare se comportă ca procesul *COPIE*. După indexarea cu un număr de sector *j* și numirea cu *mems*, sectorul *j* se comportă ca

$$\mu V. (mems.j.st\acute{a}nga?x \rightarrow mems.j.dreapta!x \rightarrow X)$$

După etichetarea multiplă cu numere naturale avem comportarea

$$\mu V. (\bigwedge_{i \geq 0} mems.j.st\acute{a}nga?x \rightarrow (\bigwedge_{i \geq 0} mems.j.dreapta!x \rightarrow X))$$

Acest proces este acum gata de a comunica în orice ocazie cu orice element al masivului de fișiere de manevră virtuale. Fiecare fișier individual respectă disciplina citirii numai din acele sectoare în care s-a scris recent.

În descrierea de mai sus, rolul numerelor naturale i și j este mai mult de a permite ca orice fișier de manevră să comunice cu orice sector de pe disc și totodată să comunice în siguranță cu utilizatorul care l-a obținut. Indicii se pot asemana de aceea cu descrierea matematică a unui comutator multiplu, care este folosit în telefonie pentru a permite conectarea între diverși abonați. Un desen simplificat este în fig. 6.2.

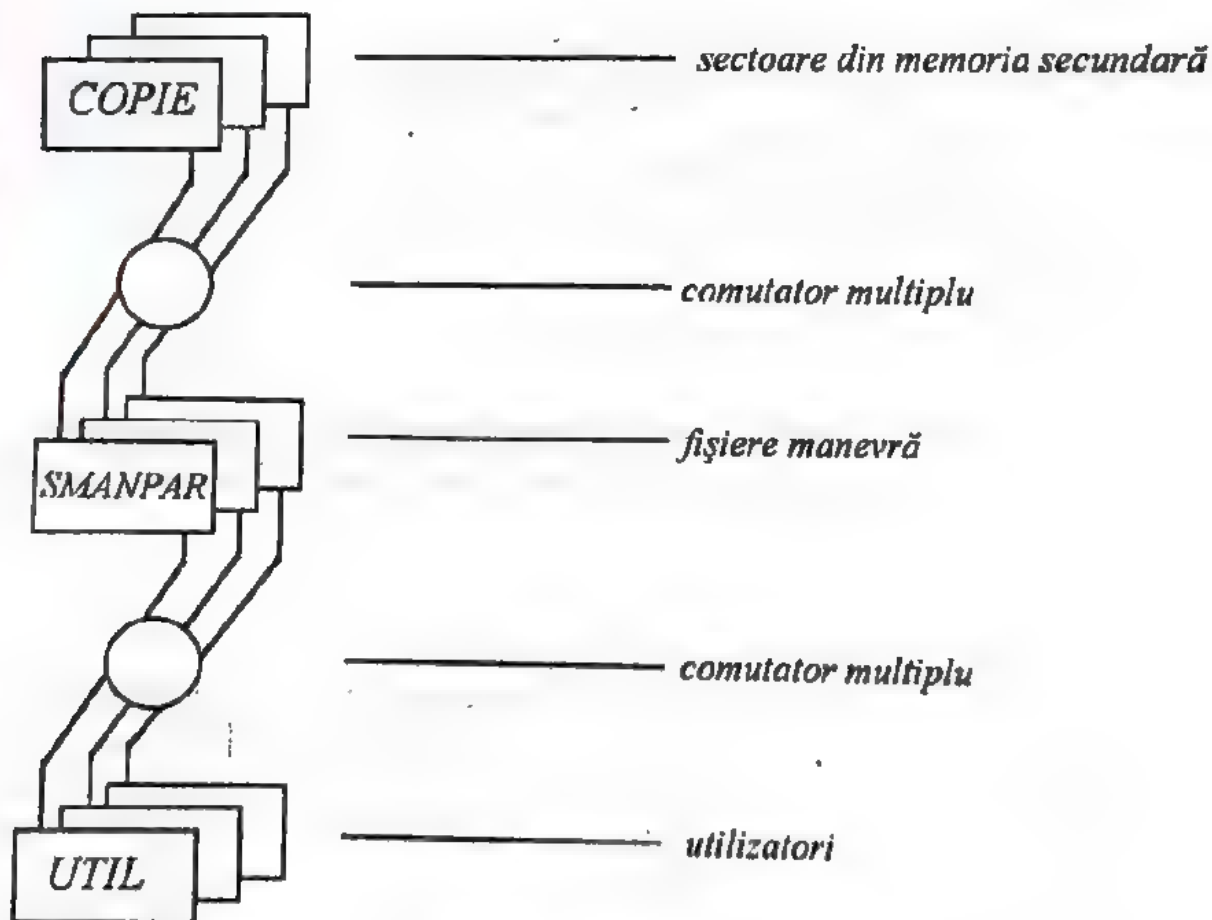


Figura 6.2

Dacă numărul sectoarelor din mediul de memorare secundar este infinit, SISFIS se comportă exact ca un masiv de fișiere de manevră simple

$$\|i:(ocupă \rightarrow (MANEV \wedge (eliberează \rightarrow STOP)))\|_{\geq 0}$$

Cu o memorie secundară de dimensiune finită există pericolul de blocaj în momentul când memoria se umple și toți utilizatorii continuă să scrie mai departe în fișierele lor de manevră. În realitate, acest risc este de fapt redus semnificativ prin punerea în așteptare a obținerii de noi fișiere când memoria este aproape plină.

6.5 Sisteme de operare

Utilizatorii unui singur calculator mare își prezintă programele pentru execuție sub formă de pachete de cartele. Datele pentru fiecare program urmează imediat. Sarcina unui sistem de operare de tip batch este de a partaja resursele calculatorului eficient între aceste *job-uri*. De aceea precizăm că programul fiecărui utilizator este executat de un program *JOB* care recepționează cartelele programului pe canalul *cr.dreapta*, execută programul cu datele aferente imediat și trimite rezultatele execuției pe canalul *lp.stânga*. Nu-i necesar să știm ceva despre structura internă a lui *JOB* – înainte el era un modul sistem tip monitor scris în *FORTRAN*. Totuși, ne bazăm pe faptul că se va termina cu succes într-un interval rezonabil de timp după startare. De aceea, alfabetul lui *JOB* este definit

$$\alpha JOB = \{cr.dreapta, lp.stânga, \sqrt{}\}$$

Dacă *LPH* reprezintă hardware-ul unei imprimante și *CRH* este hardware-ul unui cititor de cartele, o singură lucrare pentru un singur utilizator va fi executată de

$$JOB1 = (cr.CRH // p.LPH // JOB)$$

Un sistem de operare care permite rularea doar a unei singure lucrări și apoi să se termine nu prea este folosit. Cea mai simplă metodă de a partaja un calculator între mai mulți utilizatori este de a serializa executarea lucrărilor lor, una după alta

$$BATCH0 = (cr.CRH // p.LPH // *JOB)$$

Dar această proiectare ignoră câteva detalii administrative importante, ca de exemplu separarea transiterii către imprimante a fișierelor pentru fiecare lucrare și separarea citirii pachetelor de cartele conținând o lucrare de cea anterioară, astfel că o lucrare să nu poată îngloba cartelele succesoarei sale. Pentru

a rezolva aceste probleme folosim procesul LP definit în 6.2 X4 și un proces CR definit mai jos (X1)

$$\begin{aligned} JOBS &= *((cr.ocupă \rightarrow lp.ocupă \rightarrow JOB); \\ &\quad (cr.eliberează \rightarrow lp.eliberează \rightarrow SKIP)) \\ BATCH1 &= (cr.CR / lp.LP // JOBS) \end{aligned}$$

$BATCH1$ este o descriere abstractă a celui mai simplu și viabil sistem de operare, partajând un calculator între mai mulți utilizatori ale căror lucrări sunt executate rând pe rând succesiv. Sistemul de operare realizează tranziția între lucrări succesive și protejează fiecare lucrare de posibile interferențe cu predecesoarele sale.

Exemple

X1 Un cititor de cartele partajat

O cartelă specială separator este introdusă la începutul fiecărei lucrări. Cititorul de cartele este afectat pentru citirea tuturor cartelelor unei lucrări și apoi eliberat. Dacă un utilizator încearcă să citească dincolo de separator, există mai mulți separatori. Dacă utilizatorul eșuează înainte de a ajunge la separator, restul cartelelor sunt pierdute. Separatorii în plus sunt ignorați. Citirea hardware este realizată prin acțiunea $h?x$.

Cititorul de cartele partajat are nevoie să citească o cartelă în avans astfel că valoarea cartelei bufferate este folosită ca un index.

$$CR = h?x \rightarrow \text{if } x = \text{separator then } CR \text{ else } (ocupă \rightarrow CR_x)$$

unde $CR_x = (dreapta!x \rightarrow h?y \rightarrow \text{if } y \neq \text{separator then } CR_y$
 $\quad \text{else } \mu X.(dreapta!separator \rightarrow X | \text{eliberează} \rightarrow CR)$
 $\quad | \text{eliberează} \rightarrow \mu X.(h?y \rightarrow \text{if } y = \text{separator then } CR \text{ else } X))$

După ignorarea unei subsecvențe inițiale de separatori, acest proces își obține utilizatorul și copie pe canalul din dreapta secvența cartelelor diferite de separator pe care le citește de la canalul hardware. Când se detectează un separator, valoarea sa este reprodusă de atâtea ori cât este necesar ca utilizatorul să elibereze resursa. Dar dacă utilizatorul eliberează cititorul înainte să se ajungă la separator, restul cartelelor până la următorul separator trebuie citite dar ignorate. \square

Sistemul de operare $BATCH1$ este logic complet. Totuși, cu cât hardware-ul calculatorului funcționează mai repede se depășește capacitatea cititoarelor și imprimantelor de a primi la intrare și a transmite la ieșire mai multe lucrări. Pentru a stabili o legătură între intrare, ieșire și viteza de proce-

sare, este necesar și rațional să folosim două sau mai multe cititoare și imprimante. Deoarece o singură lucrare este procesată odată, cititoarele suplimentare ar putea citi în avans fișierul de cartele pentru următoarea lucrare sau lucrări și imprimantele suplimentare ar putea imprima fișierul de ieșire a lucrării anterioare. De aceea, fiecare fișier de intrare trebuie ținut într-un fișier temporar de la citirea efectivă cu cititorul și până la prelucrarea în *JOB*. Similar, fiecare fișier de ieșire trebuie reținut de la producerea lui în *JOB* și până la imprimarea efectivă pe o imprimantă. Această tehnică se numește *organizarea în coadă (spooling)*.

Structura generală a unui sistem de operare cu cozi este

$SISOP1 = sisin:COADĂIN//sisout:COADĂOUT//BATCH$

Aici *BATCH* este ca și *BATCH1* cu deosebirea că folosește subordonarea la distanță pentru a obține oricare din fișierele de intrare în așteptare și de asemenea de a obține un fișier de ieșire care este destinat imprimării următoare

$BATCH = *((cr::sisin//lp::sisout//JOB)$

Procesele care gestionează cozile de intrare și ieșire sunt definite în următoarele două exemple.

X2 Coadă de ieșire

O singură imprimantă virtuală folosește un fișier de manevră temporar (6.4 X5) pentru a reține blocurile ce trebuie imprimate, generate de procesul său utilizator. Când procesul utilizator semnalează eliberarea imprimantei virtuale, atunci o imprimantă adevărată este obținută (6.4 X3) pentru a imprima conținutul fișierului temporar.

$VLP = (temp:MANEV//$
 $\mu X.stânga?x \rightarrow temp.stânga!x \rightarrow X$
 $|eliberează \rightarrow temp.rebobinare \rightarrow$
 $(real::lp//$
 $\mu Y.(temp.dreapta?y \rightarrow real.stânga!y \rightarrow Y$
 $|temp.gol \rightarrow SKIP)))$

Tabloul nemărginit de imprimante virtuale necesare este definit

$VLPS = ||i:(ocupă \rightarrow VLP)$
 $i \geq 0$

Deoarece dorim ca imprimantele reale (6.4 X3) să fie folosite numai în modul coadă le putem declara locale sistemului ce folosește etichetare multiplă, pentru a le partaja între toate elementele masivului *VLPS* ca în 6.4 X8

$$COADĂOUT = (N:(lp:LP2)/VLPS)$$

□

X3 Coadă de intrare

Coadă de intrare este foarte asemănătoare celei de ieșire cu deosebirea că un cititor de cartele real este întâi afectat și apoi eliberat la sfârșitul intrării pentru o singură lucrare. Un proces utilizator este apoi afectat să execute lucrarea și conținutul cartelelor este transmis lui.

$$VCR = temp.MANEV$$

$$(real::cr$$

$$(\mu X.real.dreapta?x \rightarrow \text{if } x = separator \text{ then } SKIP \\ \text{else } temp.stânga!x \rightarrow X));$$

$$(temp.rebobinare \rightarrow ocupă \rightarrow$$

$$(\mu Y.(temp.dreapta?x \rightarrow dreapta!x \rightarrow Y$$

$$|temp.gol \rightarrow dreapta!separator \rightarrow Y)) \wedge (eliberează \rightarrow SKIP))$$

$$COADĂIN = (N:cr:(0:CR||1:CR)) \underset{I20}{\wedge} (||i:VCR)$$

□

Cozile de intrare și ieșire alimentează acum un număr nelimitat de cititoare și imprimante virtuale pentru folosirea de către procesul *JOB*. Ca un prim rezultat, este posibil pentru două sau mai multe procese să evolueze concurrent, partajând aceste resurse virtuale. Deoarece nu se cere vreo comunicație între lucrări, întreprinderea simplă este cea mai potrivită metodă de partajare. Această tehnică este cunoscută ca *multiprogramare*, sau dacă sunt folosite mai mult decât un procesor, este cunoscută ca *multiprocesare*. Totuși efectele logice ale multiprogramării și multiprocesării sunt aceleași. Într-adevăr, sistemul de operare definit mai jos are aceleași specificații logice ca și *SISOP1* definit mai sus.

$$SISOP = sisin:COADĂIN/sisout:COADĂOUT/BATCH4$$

unde $BATCH4 = (|||BATCH)$
 $I < 4$

Matematic vorbind, trecerea la multiprogramare s-a făcut remarcabil de simplu : din punct de vedere istoric însă a cauzat probleme mari.

În proiectarea procesului *VLP* din *COADĂOUT* (X2), procesul subordonat *MANEV* a fost folosit pentru a reține liniile pentru imprimare produse de fiecare job până când sunt imprimate efectiv pe o imprimantă reală. În ge-

neral, fişierele de ieşire sunt prea mari pentru a fi reţinute în memoria principală a unui computer, de aceea ele se reţin în memoria secundară, cum s-a arătat în 6.4 X8. Dacă toate fişierele temporare partajează aceeaşi memorie secundară, trebuie să înlocuim procesul subordonat

temp:MANEV#...

din *VLP* printr-o declaraţie a unui proces subordonat la distanţă.

temp::sisfis#...

şi apoi să declarăm sistemul (6.4 X8) ca un proces subordonat al cozii de ieşire

(sisfis:SISFIS/COADĂOUT)

Dacă volumul de date al unei cartele este semnificativ, trebuie făcută o schimbare similară la *COADĂIN*. Dacă o memorie secundară separată este disponibilă pentru acest scop, schimbarea este uşoară. Dacă nu, va trebui să partajăm aceeaşi memorie secundară între fişierele atât ale cozilor de intrare cât şi ale celor de ieşire. Aceasta înseamnă că *SISFIS* trebuie declarat ca un proces subordonat, partajat prin etichetare multiplă între cozi, aceasta implicând o schimbare în structura sistemului. Vom face această reproiectare într-o manieră *top-down*, încercând să folosim cât mai multe din modulele definite.

Sistemul de operare este compus dintr-un sistem cu multiprogramare *BATCH4* şi un sistem de intrare-ieşire servind ca proces subordonat

OP=SISTEMIO/BATCH4

Sistemul de intrare-ieşire partajează un sistem de fişiere între coada de intrare şi cea de ieşire

SISTEMIO=PART:(sisfis:SISFIS)
/(lp:COADĂOUT' || cr:COADĂIN')

şi *PART={lp.i | i ≥ 0} ∪ {cr.i | i ≥ 0}*

iar *COADĂOUT'* şi *COADĂIN'* sunt aceleaşi ca în X2 şi X3, cu deosebirea că

temp:MANEV

este înlocuit de echivalentul lui subordonat la distanţă

temp::sisfis

În proiectarea celor patru sisteme de operare descrise în acest capitol (*BATCH1*, *SISOP1*, *SISOP* și *OP*) am pus accentul mai presus de orice pe modularitate. Aceasta înseamnă că am putut refolosi mari părți din sistemele anterioare în cele ulterioare. Chiar mai important, orice decizie de detaliu este izolată în unul sau mai multe module din sistem. Ulterior, dacă trebuie schimbat un detaliu, este foarte ușor să identificăm care modul trebuie alterat și schimbările pot fi delimitate la acel modul. Printre modificările ușoare sunt

numărul imprimantelor
numărul cititoarelor de cartele
numărul de loturi concurente

Dar nu toate modificările vor fi la fel de ușoare : o schimbare a valorii cartelei separator va afecta trei module *CR (X1)*, *COADĂIN (X3)* și *JOB*.

Sunt de asemenea un număr de îmbunătățiri valorice ale sistemului care ar necesita schimbări semnificative în structura sa.

- (1) Joburile utilizator să aibă de asemenea acces la sistemul de fișiere și la dispozitivele virtuale multiple de intrare și ieșire.
- (2) Fișierele utilizatorilor să poată fi permanent reținute între joburile ce le utilizează.
- (3) Ar fi necesară o metodă de puncte de control pentru reluare rapidă în caz de cădere.
- (4) Dacă există un număr de joburi de intrare în așteptare (o listă) dar neexecutate, este nevoie de o metodă de a controla ordinea în care joburile în așteptare sunt lansate. Această chestiune este tratată mai complet în paragraful următor.

Una din problemele întâlnite când s-au făcut îmbunătățiri este imposibilitatea partajării resurselor între un proces subordonat și procesul principal corespunzător, în acele cazuri în care tehnica de etichetare multiplă nu este adecvată. Pare că o nouă definiție a subordonării este necesară, în care alfabetul procesului subordonat să nu fie o submulțime a alfabetului procesului principal. Dar aceasta este o problemă pentru o cercetare ulterioară.

6.6 Planificare

Când un număr de resurse limitate este partajat între un număr mare de potențiali utilizatori va exista totdeauna posibilitatea ca anumiți utilizatori doriți de resurse să fie nevoiți să aștepte obținerea uneia până ce un alt proces o eliberează. Dacă în momentul eliberării sunt două sau mai multe procese în

asteptarea obținerii resursei, alegerea unuia din ele este nedeterministă ca în toate exemplele date. În sine, aceasta situație nu ridică probleme. Dar să presupunem că tocmai în momentul când resursa este eliberată procesul doritor se alătură mulțimii proceselor în așteptare. Deoarece alegerea între procesele în așteptare este nedeterministă s-ar putea ca ultimul proces să fie cel ales. Dacă resursa necesită o tratare prelungită aceasta se poate întâmpla mereu. Ca rezultat, unele din procese pot fi întârziate pentru o perioadă neprevizibilă și neacceptabilă de timp. Aceasta este așa-numita problemă a *preluării infinite (overtaking)* (paragraful 2.5.5).

O soluție ar fi ca toate resursele să necesite o tratare cât mai scurtă. Aceasta se poate asigura prin multiplicarea resurselor, prin raționalizarea utilizării lor sau prin impunerea unui preț mare pentru utilizarea serviciilor asigurate. De fapt acestea sunt singurele soluții în cazul unei resurse ce necesită o tratare complexă. Din nefericire, chiar o resursă care are o medie de ocupare (încărcare) ușoară ar putea fi intens folosită pentru perioade lungi (momente de vârf sau aglomerație). Problema poate fi rezolvată prin încercarea de a netezi cererea dar aceasta nu este totdeauna posibil sau nu poate fi făcut cu succes. În timpul momentelor de vârf este inevitabil ca în medie procesele utilizator să fie supuse întârzierilor. Este important de asigurat ca aceste momente de întârziere să fie în mod rezonabil previzibile și consistente – se prefera mai mult să se știe că vei fi servit într-o oră decât să mă întreb dacă va trebui să așteptăm un minut sau o zi.

Activitatea de decizie a alocării unei resurse între mai mulți utilizatori în așteptare este cunoscută ca *planificare (scheduling)*. Pentru a planifica cu succes este necesar să știi care procese sunt în așteptarea resursei. Din această cauză, obținerea unei resurse nu mai poate fi privită ca un eveniment atomic. Ea trebuie despărțită în două evenimente

vă_rog care presupune cererea de alocare
mulțumesc care înseamnă alocarea de fapt a resursei.

Pentru orice proces, perioada dintre *vă_rog* și *mulțumesc* este perioada de așteptare a resursei. Pentru a identifica procesul care cere, vom indexa fiecare apariție a lui *vă_rog*, *mulțumesc* și *eliberează* cu un număr natural diferit. Procesul care cere o resursă o va obține de fiecare dată prin secvența corespunzătoare numărului său, aceeași construcție ca și subordonarea la distanță (6.4 X3)

$\square(\text{cer.i.vă_rog}; \text{cer.i.mulțumesc}; \dots; \text{cer.i.eliberează} \rightarrow \text{SKIP})$
 $i \geq 0$

O metodă simplă și efektivă de planificare a resurselor este de a alocă ce este disponibil procesului care a așteptat cel mai mult. Această politică este cunoscută ca *FCFS* (primul venit primul servit) sau *FIFO* (primul intrat

primul ieșit). Este o disciplină de coadă care se observă la pasagerii dintr-o coadă de autobuz.

Într-o brutărie particulară unde pâinea se face una câte una și unde consumatorii sunt în imposibilitate sau nu vor să formeze o coadă, există un mecanism alternativ pentru a realiza același efect (problema mai este cunoscută ca problema cofetăriei, patiseriei, etc. *n.t.*). S-a instalat o mașină care generează tichete cu o ordine serială ascendentă strictă în ce privește numerele. La intrarea în brutărie un cumpărător primește un tichet. Când un lucrător este gata, el cheamă cumpărătorul cu cel mai mic număr de tichet care n-a fost încă servit. Presupunem că până la C cumpărători pot fi serviți simultan.

Exemplu

X1 Algoritmul brutăriei

Trebuie să ținem următoarele socoteli

r - clienți care au spus *vă rog*
 m - clienți care au spus *mulțumesc*
 e - clienți care au eliberat resursele

În mod clar, totdeauna $e \leq m \leq r$. De asemenea, totdeauna r va fi numărul dat următorului client care intră în brutărie și m este numărul următorului client de servit. Mai mult, $r-m$ este numărul clienților în așteptare și $C+e-m$ este numărul lucrătorilor în așteptare. Toți contorii sunt inițial 0 și pot reveni la 0 oricând sunt egali – de exemplu noaptea după ce a plecat ultimul client.

Una din sarcinile principale ale algoritmului este să asigure că nu există simultan o resursă liberă și un client așteptând. Ori de câte ori este o astfel de situație, următorul eveniment trebuie să fie *mulțumesc* de la un client obținând resursa

```
BRUTĂRIE =  $B_{0,0,0}$ 
 $B_{r,m,e}$  = if  $0 < e = m = r$  then BRUTĂRIE
           else if  $C + e - m > 0 \wedge r - m > 0$ 
             then  $m.mulțumesc \rightarrow B_{r,m+1,e}$ 
             else  $(r.vă\_rog \rightarrow B_{r+1,m,e}$ 
                  |  $(\square i.eliberează \rightarrow B_{r,m,e+1}))$ 
                   $i < m$ 
```

Algoritmul brutăriei se datorează lui Leslie Lamport.

□

7 Discuție

7.1 Introducere

Principalul obiectiv al acestei lucrări din domeniul proceselor comunicante a fost găsirea celei mai simple posibile teorii matematice cu următoarele proprietăți :

- (1) Să poată descrie o gamă cât mai largă de aplicații pe calculator, de la automate de vândut, controlul proceselor și simularea cu evenimente discrete, până la sisteme de operare cu resurse partajabile.
- (2) Să fie capabilă de o implementare eficientă pe o varietate de arhitecturi convenționale și de ultimă oră, de la calculatoare cu time-sharing, multiprocesoare, până la rețele de microprocesoare ce comunică între ele.
- (3) Să asigure un sprijin clar programatorului în sarcinile sale de specificare, proiectare, implementare, verificare și validare a unor sisteme complexe.

Este evident că nu putem cere ca toate aceste obiective să fie realizate de o manieră optimă. Există însă totdeauna speranța că o metodă diferită radical de cele existente sau câteva schimbări importante în detalierea definițiilor ne-ar conduce la un succes mai mare corespunzător unuia sau mai multor din obiectivele enumerate mai sus. Acest capitol încearcă o discuție a unor alternative explorate de autor împreună cu alții și o explicație de ce această teorie și nu alta. De asemenea, dă ocazia autorului să recunoască influența cercetării originale a altor savanți în domeniu. În fine, se speră încurajarea cercetării ulterioare privind bazele acestei teorii precum și larga sa aplicabilitate practică.

7.2 Partajarea memoriei

Primele propuneri din 1960 privind programarea concurentă a activităților într-un calculator provin natural din dezvoltările contemporane ale arhitecturii

calculatoarelor și sistemelor de operare. În acele timpuri puterea de procesare era insuficientă și scumpă și era considerat pierdere de timp ca un procesor să aștepte comunicația cu un echipament periferic mai lent sau chiar cu utilizatorul. De aceea, ulterior s-au dezvoltat controllere speciale (procesoare de canal) pentru operații de I/O independente permițând astfel procesorului central să se angajeze în alte acțiuni. Pentru a ține ocupat procesorul central, un sistem de operare cu partajarea timpului trebuia să asigure existența simultană a unor programe în memoria principală a calculatorului. Astfel, în orice moment orice program putea accesa controllere de I/O în timp ce un alt program putea folosi procesorul principal. La terminarea operației de I/O o întrerupere permitea sistemului de operare reconsiderarea programului ce a generat accesul astfel ca acesta să se bucure în continuare de atenția procesorului central.

Schema descrisă mai sus sugerează că procesorul central și toate procesoarele de canal ar trebui conectate la memoria principală a calculatorului determinând ca accesele la memorie să fie întrepesute. Cu toate acestea, fiecare program executat era de obicei o lucrare completă afectată unui utilizator anume și complet independentă de alte lucrări.

Din acest motiv exista o mare grijă în proiectarea hardware-ului și software-ului pentru împărțirea memoriei în segmente disjuncte, unul pentru fiecare program, cu asigurarea că nici un program nu putea interfera cu altul. Când a devenit posibil de atașat mai multe procesoare independente în același calculator efectul a fost creșterea răspunsului lucrărilor (joburilor). Dacă sistemele de operare originale erau bine structurate acest lucru se putea realiza cu puține schimbări în sistemul de operare și chiar și mai puține în programele pentru lucrările ce se executau.

Dezavantajele partajării unui calculator între mai multe lucrări distincte erau

- (1) Dimensiunea memoriei necesare creștea liniar cu numărul de lucrări executate simultan
- (2) Timpul necesar fiecărui utilizator pentru obținerea rezultatelor la lucrarea sa creștea de asemenea cu excepția lucrărilor de cea mai mare prioritate.

De aceea pare rezonabil de a permite unei singure lucrări să profite de paralelismul asigurat de hardware-ul calculatorului prin inițierea mai multor procese concurente în zona de memorie alocată unui singur program.

7.2.1 Multiplicarea firelor de execuție

Prima propunere de acest fel s-a bazat pe un salt (comandă `go to`). Dacă L este o etichetă din program, comanda

fork L

transferă controlul etichetei *L* și de asemenea permite executarea următoarei comenzi din secvență. Din acel moment efectul este ca și cum două procesoare execută același program în același timp. Fiecare își menține însă firul său de execuție. Deoarece fiecare fir de control poate produce din nou un *fork* această tehnică de programare este cunoscută ca *multiplicarea firelor de execuție*.

Odată dovedită o metodă ca un proces să poată fi descompus în două este necesară o metodă ca două procese să poată fi unificate. O propunere simplă este asigurarea unei comenzi

join

care poate fi realizată numai când două procese o execută simultan. Primul proces care ajunge la comandă trebuie mai întâi să aștepte până ce un altul ajunge de asemenea. După aceea cele două devin unul singur.

Datorită generalității sale, multiplicarea firelor de execuție este o tehnică incredibil de complexă și predispusă la erori, nefiind recomandată decât în programe mici. Ca o scuză ea a fost inventată înainte de a apare programarea structurată, când chiar FORTRAN-ul era considerat un limbaj de programare de nivel înalt !

O variantă a comenzii *fork* este încă utilizată în sistemul de operare UNIX™. *Fork*-ul nu folosește nici o etichetă. Efectul ei este de a face o copie a întregii memorii alocate programului, copia devenind un nou proces. Atât procesul original cât și copia își continuă execuția cu următoarea comandă de după *fork*. Este asigurată de asemenea o facilitare pentru fiecare program de a descoperi dacă este *părinte* sau *fiu*. Alocarea de zone disjuncte de memorie proceselor elimină principalele dificultăți și pericole ale tehnicii multiplicării firelor de execuție dar poate fi inefficientă atât în timp cât și în spațiu. Aceasta înseamnă că este permisă concurența numai la nivelul global al unui job iar utilizarea sa la o scară mai mică este descurajată.

7.2.2 Cobegin ... coend

O soluție la problema multiplicării firelor de execuție a propus-o E. W. Dijkstra prin asigurarea că după un *fork* cele două procesoare execută blocuri de program complet diferite fără posibilitatea saltului între ele. Dacă *P* și *Q* sunt astfel de blocuri, comanda compusă

cobegin *P*, *Q* coend

determină ca *P* și *Q* să pornească simultan și să se execute concurent până ce ambele s-au terminat. După aceasta rămâne numai un singur procesor să exe-

cute mai departe alte comenzi. Această comandă structurată poate fi implementată de comenzile nestructurate *fork* și *join*, folosind etichetele *L* și *J*

fork *L*; *P*; go to *J*; *L*: *Q*; *J*: join

Generalizarea la mai mult de două procese componente este imediată și evidentă

cobegin *P*; *Q*; ... ;*R* coend

Un mare avantaj al acestei notații structurate este ușurința de a înțelege ce se întâmplă de fapt, în special dacă variabilele folosite în fiecare din blocuri sunt distincte de variabilele folosite în altele (o restricție care poate fi verificată sau impusă de un compilator pentru un limbaj de nivel înalt). În acest caz procesele se spune că sunt *disjuncte* și (în absența comunicației) execuția concurrentă a lui *P* și *Q* are exact același efect ca și execuția lor secvențială în altă ordine

begin *P*; *Q* end = begin *Q*; *P* end = cobegin *P*; *Q* coend

Mai mult, metodele de demonstrare pentru stabilirea corectitudinii compunerii paralele pot fi chiar mai simple decât cazul secvențial. Din această cauză propunerea lui Dijkstra este baza construcției paralele din această carte. Principala schimbare este notația. Pentru a evita confuzia cu compunerea paralelă eu am introdus între procese operatorul **||** permițându-se astfel utilizarea parantezelor pentru includerea comenzilor în loc de scrierea mai stânjenitoare **cobegin ... coend**.

7.2.3 Regiuni critice condiționale

Restricția ca procesele concurente să nu partajeze variabile are drept consecință faptul că ele nu pot comunica sau interacționa între ele în vreun fel, o restricție care reduce substanțial valoarea potențială a concurenței.

După citirea acestei cărți introducerea canalelor de intrare și ieșire (simulate) poate părea soluția evidentă. La începuturi o tehnică evidentă (sugerată de hardware-ul calculatoarelor) era de a comunica prin partajarea memoriei principale între procesele concurente. Dijkstra a arătat cum se putea realiza aceasta relativ fără griji cu ajutorul regiunilor critice (paragraful 6.3.) protejate de semafoare de excludere mutuală. Eu am propus mai târziu ca această metodă să fie formalizată prin notațiile unui limbaj de nivel înalt. Un grup de variabile care trebuie actualizate în regiunile critice din mai multe procese partajante s-au declarat ca resurse partajate, de exemplu


```
shared n : integer;
shared poziție : record x, y : real end
```

Fiecare regiune critică care actualizează această variabilă este precedată de o clauză *with*, indicând numele variabilei

```
with n do n := n + 1;
with poziție do begin x := x + deltax; y := y + deltay end
```

Avantajul acestei notații este acela că un compilator introduce automat semafoarele necesare și delimitează fiecare regiune critică prin operațiile necesare *P* și *V*. Mai mult, poate verifica la compilare dacă vreo variabilă partajată poate fi accesată sau actualizată într-o regiune critică protejată de semafoarele corespunzătoare.

Cooperarea între procesele concurente care partajează memorie necesită de multe ori alte forme de sincronizare. De exemplu, un proces actualizează o variabilă în ideea că alte procese ar putea citi noua variabilă. Celelalte procese nu trebuie să citească variabila până ce actualizarea n-a avut loc. Similar, primul proces nu trebuie să actualizeze variabila până ce celelalte procese n-au citit versiunea anterioară.

Pentru a rezolva această problemă o facilitare convenabilă este oferită prin regiunea critică condițională. Aceasta ia forma

```
with var_part when condiție do regiune critică
```

La intrarea în regiunea critică este testată valoarea condiției. Dacă este adevărată, regiunea critică este executată normal. Dacă condiția este falsă această intrare în regiunea critică este amânată astfel că altor procese li se permite intrarea în regiunile lor critice și actualizarea variabilei partajate. La terminarea actualizărilor condiția este retestată. Dacă a devenit adevărată, procesul amânat poate pătrunde în regiunea sa critică, altfel procesul este suspendat din nou, iar dacă trebuie făcută o alegere între mai mult de un proces amânat, alegerea este arbitrară.

Pentru a rezolva problema actualizării și citirea mesajului de mai multe procese se declară ca făcând parte din resursă o variabilă întregă, contor pentru un număr de procese care trebuie să citească mesajul înainte ca el să fie actualizat din nou

```
shared mesaj : record contor : integer; conținut : ... end;
mesaj.contor := 0;
```

Procesul care actualizează conține o regiune critică de formă

```

with mesaj when contor = 0 do
  begin conținut := ... ;
    ...;
    contor := număr de cititori
  end

```

Fiecare proces cititor conține o regiune critică de forma

```

with mesaj when contor > 0 do
  begin copia personală := conținut, contor := contor-1 end

```

Regiunile critice condiționale pot fi implementate cu ajutorul semafoarelor. Comparativ cu utilizarea directă a semafoarelor de sincronizare de către programator, plusul adus de regiunile critice condiționale este destul de mare deoarece condițiile tuturor proceselor în așteptare la intrarea în regiunea critică trebuie retestate la fiecare ieșire din regiune. Din fericire, condițiile nu trebuie testate mai frecvent decât după o ieșire deoarece restricțiile impuse accesului la variabilele partajate ne asigură că o condiție testată de un proces în așteptare, poate fi schimbată ca valoare numai dacă variabila partajată însăși își schimbă valoarea. Toate celelalte variabile din condiție trebuie să fie particulare procesului în așteptare care evident nu le poate schimba când este în așteptare.

7.2.4 Monitoare

Dezvoltarea monitoarelor s-a inspirat din ideea de *clasă* (*class*) din SIMULA 67, care la rândul ei era o generalizare a conceptului de procedură din ALGOL 60. Principala lor caracteristică este aceea că toate operațiile importante cu date (incluzând inițializarea) ar putea fi reunite împreună într-o declarație de structură și tip de dată. Aceste operații ar putea fi invocate prin apeluri de procedură ori de câte ori sunt necesare proceselor care partajează datele. Caracteristica importantă a unui monitor este că numai una din procedurile corpului său poate fi activă la un moment dat. Atunci când două procese apelează simultan o procedură (aceeași procedură sau diferite), unul din apeluri este întârziat până când celălalt este terminat. Astfel corpurile procedurilor joacă rolul regiunilor critice condiționale protejate de același semafor. De exemplu, un semafor foarte simplu se comportă ca o variabilă *contor*. În notația PASCAL PLUS avem

```

1 monitor contor;
2 var n : integer;
3 procedure *incr, begin n := n + 1 end;
4 procedure *decr, when n > 0 do begin n := n - 1 end;

```

```

5 function *zero : Boolean; begin zero := (n = 0) end;
6 begin n := 0;
7   ...;
8   if n ≠ 0 then print (n)
9 end

```

- Linia 1 declară monitorul și îi dă numele *contor*.
 2 declară variabila partajată *n* locală monitorului. Ea este inaccesibilă în afara monitorului.
 3 } { declară trei proceduri cu corpurile lor.
 4 } { Asteriscul ne asigură că ele pot fi apelate
 5 } { numai din programul care utilizează monitorul.
 6 Monitorul pornește execuția aici.
 7 Cele trei puncte-puncte reprezintă instrucțiuni incluse în blocul care utilizează monitorul.
 8 Valoarea finală a lui *n* (dacă $\neq 0$) este afișată la ieșirea din blocul utilizator.

O nouă instanță a acestui monitor poate fi declarată local unui bloc *P*

instance rachetă : contor, P

În interiorul blocului *P*, procedurile care încep cu * pot fi apelate prin comenzi

rachetă.incr, ... rachetă.decr, ... ; if rachetă.zero then ...

Totuși o procedură fără * sau variabilă, ca de exemplu *n*, nu pot fi accesate din interiorul lui *P* și supravegherea acestei restricții este controlată de compilator. Excluderea mutuală inerentă monitorului ne asigură că o procedură a sa poate fi apelată în siguranță de orice număr de procese din *P* și nu există pericolul interferării actualizării lui *n*. De observat că o încercare de a apela *rachetă.decr* când *n* = 0 va fi întârziată până ce alt proces din *P* apelează *rachetă.incr*. Aceasta ne asigură că valoarea lui *n* nu poate fi negativă nicio dată.

Efectul declarării unei instanțe a monitorului este explicat printr-o variantă a regulii de copiere a unui apel de procedură în ALGOL 60. Mai întâi, se face o copie a textului monitorului, blocul utilizator. Blocul utilizator *P* este copiat în locul celor trei puncte-puncte din monitor și toate numele locale ale monitorului sunt prefixate cu numele instanței, cum se arată mai jos


```

rachetă.n : integer;
procedure rachetă.incr; begin rachetă.n := rachetă.n + 1 end;
procedure rachetă.decr;
  when rachetă.n > 0 do begin rachetă.n := rachetă.n - 1 end;
function rachetă.zero : Boolean;
  begin rachetă.zero := (rachetă.n = 0) end
begin rachetă.n := 0;
  P;
  if rachetă.n ≠ 0 then print (rachetă.n)
end

```

De observat cum regula de copiere a făcut imposibil pentru procesul utilizator să uite inițializarea valorii lui *n* sau să uite imprimarea valorii sale finale când este necesar.

Ineficiența testării repetate a condițiilor de intrare au dus la proiectarea monitoarelor după niște scheme mai elaborate pentru așteptarea explicită și semnalarea explicită a reluării proceselor în așteptare. Aceste scheme permit chiar unui apel de procedură suspendarea în mijlocul execuției sale (pe o condiție), astfel că după eliberarea automată a excluderii un apel ulterior de procedură făcut de un alt proces poate declanșa reluarea procesului suspendat. În acest fel, un număr de tehnici ingenioase de planificare pot fi eficient implementate, și totodată, acum cred că o complexitate suplimentară este pe deplin justificată.

7.2.5 Monitoare imbricate

O instanță a unui monitor poate fi privită ca un semafor pentru a proteja o singură resursă ca de exemplu o imprimantă care nu trebuie folosită de mai mult de un proces odată. Un astfel de monitor ar putea fi declarat

```

monitor resursă;
var liber : Boolean;
procedure *capturează; when liber do liber := fals;
procedure *eliberează; begin liber := adevărat end;
begin liber := adevărat, ... end

```

Totuși, protecția realizată de acest monitor poate fi ocolită de un proces care utilizează resursă fără capturarea ei sau compromisă de altul care uită să o elibereze. Ambele pericole pot fi evitate printr-o construcție similară resursei virtuale (6.4 X4). Aceasta ia forma unui monitor declarat local într-un monitor – resursă prezentat mai sus. Numele resursei virtuale este notat cu * pentru a o face accesibilă declarării de către procesele utilizator. Totuși * este eliminat

din numele procedurilor **capturează* și **eliberează* astfel că acestea pot fi folosite numai în interiorul monitorului resursei virtuale, neputând fi folosite abuziv de alte procese

```
monitor resursă;
  liber : Boolean;
  procedure capturează;
    when liber do liber := fals;
  procedure capturează;
    begin liber := true end
  monitor *virtual;
    procedure *utilizează (l:linie); begin ... end;
    begin capturează; ... ; eliberează end
begin liber := adevărat; ... end
```

O instanță a acestui monitor este declarată astfel

```
instance imprimantă : resursă; P
```

Un bloc din *P* care cere transmisia unui fișier la o imprimantă se scrie

```
instance prog : imprimantă.virtual;
begin ... prog.utilizează (l1); ... prog.utilizează (l2); ... end
```

Operațiile necesare de capturare și eliberare a imprimantei sunt automat inserate de monitorul virtual înainte și după acest bloc utilizator într-o manieră care previne utilizarea antisocială a imprimantei. În principiu ar fi posibil pentru blocul utilizator să se împartă în mai multe procese paralele toate utilizând instanța *prog* din monitorul virtual dar probabil că nu este aceasta intenția. Un monitor care trebuie folosit numai pentru un singur proces este cunoscut în PASCAL PLUS ca un *plic* și poate fi implementat mai eficient fără excludere sau sincronizare. Compilatorul verifică dacă nu este din neatenție partajat.

Semnificația acestor declarații de instanțe poate fi înțeleasă prin aplicarea repetată a regulii de copiere cu rezultatul arătat în exemplul alăturat.

Copierea explicită arătată aici este numai pentru cei mai puțin obișnuiți cu această tehnică. Un programator mai experimentat n-ar dori niciodată să vadă versiunea expandată sau chiar să se gândească la ea.

Aceste notații au fost folosite în 1975 pentru descrierea unui sistem de operare similar cu acela din paragraful 6.5. Ele au fost implementate apoi în PASCAL PLUS. Efecte extrem de ingenioase pot fi obținute prin mixarea *** cu imbricarea.

```

var imprimantă.liber : Boolean;
procedure imprimantă.capturează;
    when imprimantă.liber do imprimantă.liber := fals;
procedure imprimantă.eliberează; begin imprimantă.liber := adevărat
                                end;

begin imprimantă.liber := adevărat
    .
    .
    .
begin
    procedure prog.imprimantă.lutimizează (l : linie); begin ... end;
    imprimantă.acaparează;
    ... prog.imprimantă.lutimizează (l1);
    ... prog.imprimantă.lutimizează (l2);
    imprimantă.eliberează;
end;
    .
    .
    .
end

```

Cu toate acestea, notațiile din PASCAL și SIMULA par puțin stângace și explicațiile în termenii substituției și redenumirii sunt puțin cam greu de urmărit. Datorită criticilor lui Edsger W. Dijkstra asupra acestor aspecte am fost impulsionat să proiectez procesele comunicante secvențiale.

Totuși, este clar din paragraful 6.5. că partajarea ca concept implică complicații cu toate că este exprimată în cadrul teoretic al proceselor comunicante sau în cadrul regulii de copiere și semanticii apelurilor de procedură din PASCAL PLUS. Alegerea între limbaje pare a fi în parte o chestiune de gust sau eficiență. Pentru implementarea unui sistem de operare pe un calculator cu memorie principală partajată, PASCAL PLUS are probabil un avantaj.

7.2.6 Ada™

Facilitățile oferite pentru programarea concurentă în Ada sunt un amalgam de apel de procedură la distanță din PASCAL PLUS cu forma mai puțin structurată de comunicare prin intrare și ieșire. Procesele se cheamă task-uri și ele comunică printr-o declarație *call* (ca și apelurile de procedură cu parametri de ieșire și intrare) și declarații *accept* (care sunt atât în forma lor sintactică cât și în efect ca declarațiile de procedură). O declarație tipică de *accept* ar fi

```
accept scrie (V : in integer, ANTER : out integer) do
  ANTER := K; K := V end
```

Un apel corespunzător ar putea fi

```
scrie(37,X)
```

Identificatorul *scrie* este cunoscut ca *nume de intrare*.

O declarație *accept* și una *call* cu același nume în taskuri diferite sunt executate când ambele procese sunt gata să le execute împreună. Efectul este precum urmează

- (1) Parametrii de intrare sunt copiați prin apel la procesul care-i acceptă.
- (2) Este executat corpul declarației *accept*.
- (3) Valorile parametrilor de ieșire sunt copiați înapoi prin apel.
- (4) Apoi ambele taskuri își continuă execuția cu următoarele lor declarații.

Execuția corpului unui *accept* este cunoscută ca tehnica de *rendezvous*, deoarece atât taskul apelant cât și cel apelat pot fi gândite că se execută împreună. *Rendezvous*-ul este o calitate atrăgătoare pentru Ada deoarece simplifică practica foarte utilizată a alternării ieșirii cu intrarea fără complicarea cazului când se folosește numai intrarea sau numai ieșirea.

Analogul lui Ada pentru □ este declarația *select* care ia forma

```
select
  accept citește (v : out integer) do v := B[i] end; i := i + 1; ...
  or accept scrie (v : in integer) do B[j] := v end; j := j + 1; ...
  or ...
end select
```

Exact una din alternativele separate de *or* vor fi selectate pentru execuție, depinzând de alegerea făcută de task-ul (-urile) apelant(e). Declarațiile rămase după *end* în alternativa selectată din *accept* sunt executate la terminarea *rendezvous*-ului, concurrent cu derularea taskului apelant. Selectarea unei alternative poate fi inhibată printr-o condiție *when* falsă, de exemplu

```
when not liber => accept ...
```

Se realizează astfel efectul unei regiuni critice condiționale.

Una din alternativele dintr-o declarație *select* poate începe cu un *delay* în loc de un *accept*. Această alternativă poate fi selectată dacă nici o alta nu

este selectată pe o perioadă mai mare decât un număr specificat de secunde. Scopul este de a ne proteja de pericolul ca erori hardware sau software să determine amânarea continuă a declarației select. Deoarece modelul nostru matematic face deliberat abstracție de timp, o întârziere nu poate fi reprezentată adecvat decât cu excepția cazului când se permite selecția complet nedeterministă a alternativei, începând cu întârzierea.

Una din alternative într-o declarație *select* poate fi *terminate*. Această alternativă este selectată când toate taskurile care ar putea apela taskul dat s-au terminat. De asemenea s-a terminat și taskul dat. Aceasta nu este o situație la fel de convenabilă ca o declarație interioară din PASCAL PLUS care permitea monitorului să pună toate lucrurile în ordine la terminare.

O declarație *select* poate avea o clauză *else* care este selectată fie din cauză că nici una din celelalte alternative nu poate fi selectată imediat fie din cauză că toate condițiile *when* sunt false fie din cauză că nu există un apel corespunzător în așteptare în alt task. Aceasta ar putea fi ceva echivalent unei alternative cu *delay 0*.

Un apel *call* poate fi de asemenea protejat împotriva întârzierilor arbitrare printr-o declarație *delay* sau o clauză *else*. Aceasta ar putea conduce la ineficiență în implementare într-o rețea distribuită de procesoare.

Taskurile în Ada sunt declarate aproape la fel ca procesele subordonate din paragraful 4.5. La fel ca monitoarele din PASCAL PLUS, fiecare task poate deservi orice număr de procese apelante. Mai mult, programatorul trebuie să se îngrijească ca taskul să se termine normal. Definiția unui task este compusă din două părți, specificarea și corpul. Specificarea dă taskului numele său precum și numele și tipul parametrilor tuturor intrărilor prin care taskul poate fi apelat. Aceasta este informația utilă programatorului și folosită de compilatorul respectiv. Corpul taskului definește comportarea sa și poate fi compilat separat de programul apelant.

Fiecărui task în Ada îi poate fi atribuit o *prioritate* fixă. Dacă există mai multe taskuri decât procesoare, taskurile cu prioritate mai scăzută vor fi neglijate. Prioritatea în execuția unui *rendezvous* este mai mare decât prioritățile taskurilor apelante sau apelate. Indicatorul de prioritate se numește *pragma*. Prin acest indicator se intenționează îmbunătățirea timpilor critici de răspuns comparați cu cei necritici și nu se intenționează afectarea comportării logice a programului. Aceasta este o idee excelentă deoarece separă preocuparea pentru corectitudinea logică abstractă de problematica răspunsului în timp real, care în general poate fi mai ușor de rezolvat printr-o judicioasă selecție a hardware-ului sau prin experimente.

Ada oferă un număr de facilități adiționale. Astfel, este posibil de testat câte apeluri așteaptă la o intrare a unui monitor. Un task poate termina brusc alt task (*abort*) și toate taskurile depinzând de el. Taskurile pot accesa și actualiza variabile partajate. Aceste efecte pot fi făcute chiar mai imprevizibile prin faptul că compilatoarelor le sunt permise întârzierile, reordonările sau în-

trețeserea actualizărilor ca și cum variabila n -ar fi fost partajată. Există de asemenea alte efecte de interacțiune și complexități adiționale cu alte caracteristici pe care nu le-am menționat.

În afara complexităților arătate în paragraful precedent, multitasking-ul în Ada pare a fi mai bine proiectat pentru implementare și utilizare pe un multiprocesor cu memorie partajată.

7.3 Comunicarea

Explorarea posibilităților de structurare a unui program sub formă de rețea de procese comunicante a fost printre altele motivată de dezvoltarea spectaculoasă a hardware-ului. Nașterea microprocesorului a redus rapid costul puterii de procesare cu câteva ordine de mărime. Totuși, puterea fiecărui microprocesor individual era mult mai mică decât a unui calculator standard tradițional și într-un fel scump. A apărut astfel a fi mult mai economic obținerea unei puteri mărite prin utilizarea mai multor microprocesoare care să coopereze la același task. Aceste microprocesoare puteau fi conectate ieftin prin canale de-a lungul cărora să comunice între ele. Fiecare microprocesor putea avea memoria sa principală locală pe care să o acceseze la viteză maximă, evitând astfel blocajele ce ar rezulta când mai multe microprocesoare accesează la un moment dat o memorie partajată.

7.3.1 Conducute

Cel mai simplu model de comunicație între elemente de procesare este transferul unidirecțional al mesajului între fiecare proces și vecinul său, ceea ce constituie o conductă (*pipe*) liniară ca în paragraful 4.4. Ideea a fost prima dată pusă în discuție de Conway care a ilustrat-o prin exemple similare celor din 4.4 X2 și X3 cu deosebirea că se aștepta ca cele două procese componente ale conductei să se termine cu succes în loc să cicleze infinit. El a propus modalitatea de a fi folosită structura de conductă pentru scrierea unui compilator în mai mulți pași pentru un limbaj de programare. Astfel, pe un calculator cu o memorie principală adecvată, toți pașii sunt activi simultan în memorie și controlul este trecut pe rând între pași împreună cu transferul de mesaje, simulând astfel execuția paralelă. Pe un calculator cu mai puțină memorie principală, numai un pas este activ o dată el trimițându-și ieșirea într-un fișier din memoria secundară. La terminarea fiecărui pas, începe următorul pas care își preia intrarea din fișierul produs de predecesorul său. Totuși, rezultatul final al compilatorului este exact același în ciuda diferențelor radicale în problema execuției. Este caracteristic pentru abstractizarea cu succes în programare că implementarea poate fi făcută prin mai multe metode care la rândul lor sunt eficiente funcție de diferite circumstanțe. În acest caz sugestia lui

Conway ar putea fi foarte prețioasă pentru implementări software pe o gamă de calculatoare, oferind o gamă largă de opțiuni pentru mărimea memoriei.

Conceptul de conductă este de asemenea metoda standard de comunicație în sistemul de operare UNIX™, unde notația "|" corespunde lui ">" de la noi.

7.3.2 Canale bufferate multiple

Modelul conductei permite unui lanț liniar de procese să comunice numai într-o singură direcție și nu contează dacă secvența de mesaje este bufferată sau nu. Generalizarea naturală a conductei este de a permite fiecărui proces să comunice cu orice alt proces în orice direcție. La prima vedere, pare aproape natural să asigurăm în aceste condiții bufferarea pe toate canalele de comunicație. În proiectarea sistemului de operare RC4000, facilitatea de comunicare bufferată a fost implementată în kernel și a fost utilizată pentru comunicarea între module asigurând servicii către un nivel superior. La o scară mai mare, o rețea de memorare și comutare a pachetelor, ca de exemplu ARPAnet în USA, interpune inevitabil buffere între sursa și destinația mesajelor.

Când modelul de comunicație între procese este generalizat de la un lanț liniar la o rețea care poate fi și ciclică, prezența sau absența bufferării poate determina o diferență vitală privind comportarea logică a sistemului. Prezența bufferării nu este totdeauna favorabilă: de exemplu, este posibil de a scrie un program care se poate bloca dacă lungimea bufferului este permis să depășească cinci, tot așa cum un program diferit se va bloca dacă lungimea bufferului nu este permis să depășească cinci. Pentru a evita astfel de iregularități, lungimea tuturor bufferelor ar putea fi infinită. Din nefericire, aceasta ne conduce la grave probleme de eficiența implementării când memoria principală este umplută cu buffere de mesaje. Tratarea matematică este de asemenea complicată prin faptul că orice rețea este un automat cu un număr infinit de stări, chiar când procesele componente sunt în număr finit. În fine, pentru o rapidă și controlabilă interacțiune între oameni și calculatoare, bufferele trebuie afectate numai canalelor, deoarece presupun o întârziere între stimul și răspuns. Dacă ceva nu e în regulă în procesarea unui stimul bufferat, este mult mai dificil de urmărit eroarea și de eliminat. Bufferarea este o tehnică de procesare de tip batch și ar trebui evitată ori de câte ori interacțiuni rapide sunt mai importante decât utilizarea intensivă a unui procesor.

7.3.3 Multiprocesare funcțională

Un proces determinist poate fi definit în termenii unei funcții matematice de la canalele sale de intrare la cele de ieșire. Fiecare canal este identificat cu secvența infinit extensibilă de mesaje care trece de-a lungul lui. Astfel de funcții sunt definite în general recursiv folosind structura secvențelor de in-

trare, mai puțin cazul unei secvențe de intrare vide. De exemplu, procesul care emite rezultatul multiplicării fiecărui număr de intrare cu un n este definit

$$prod_n(stânga) = \langle n \times stânga_0 \rangle \wedge prod_n(stânga')$$

O funcție care are două șiruri sortate (strict) ca parametri și emite șirul comun sortat (strict) este definită

$$\begin{aligned} dup2(stg1, stg2) = \\ \text{if } stg1_0 < stg2_0 \text{ then } \langle stg1_0 \rangle \wedge dup2(stg1', stg2) \\ \text{else if } stg2_0 < stg1_0 \text{ then } \langle stg2_0 \rangle \wedge dup2(stg1, stg2') \\ \text{else } \langle stg2_0 \rangle \wedge dup2(stg1', stg2') \end{aligned}$$

O rețea neciclică poate fi reprezentată printr-o compunere de astfel de funcții. De exemplu, o funcție care reunește trei șiruri sortate de intrare poate fi definită ca

$$dup3(stg1, stg2, stg3) = dup2(stg1, dup2(stg2, stg3))$$

Figura 7.1 prezintă diagrama de conexiune a acestei funcții.

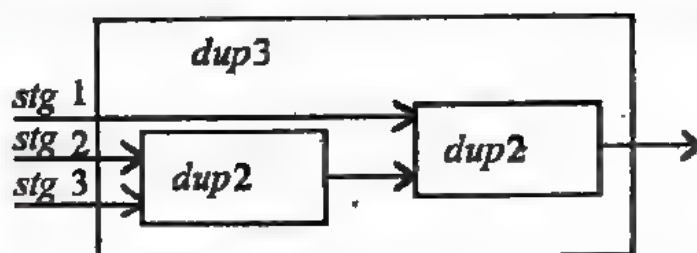


Figura 7.1

O rețea ciclică poate fi construită cu ajutorul unui set de ecuații recursive mutual. De exemplu, considerăm problema atribuită de Dijkstra lui Hamming și anume de a defini o funcție care generează într-o secvență crescătoare toate numerele care au numai pe 2, 3 și 5 ca factori netriviali. Primul astfel de număr este 1. Dacă x este un astfel de număr sunt și $2 \times x$, $3 \times x$ și $5 \times x$ de asemenea. Pentru problema noastră utilizăm trei procese $prod_2$, $prod_3$ și $prod_5$ pentru a genera aceste produse, rezultatele alimentând procesul $dup3$, care ne asigură că sunt sortate într-o ordine crescătoare (fig. 7.2).

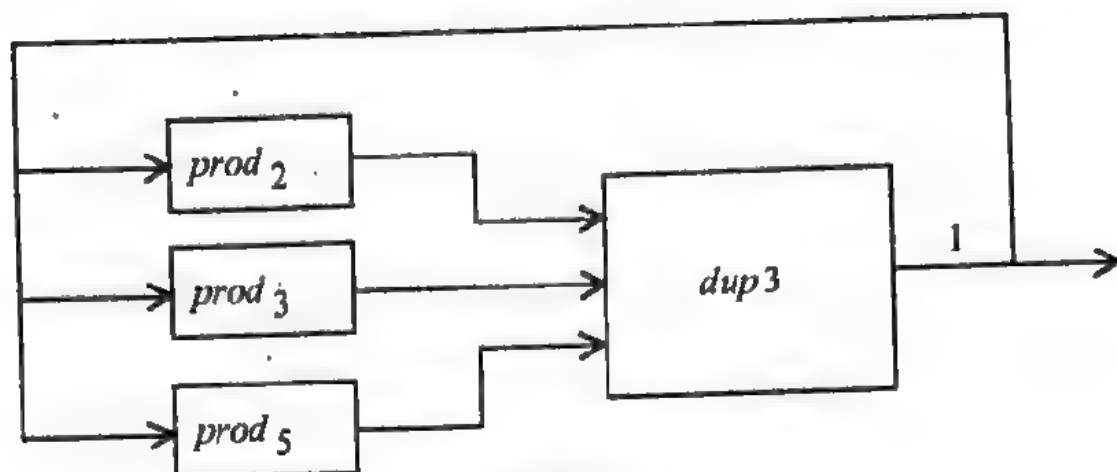


Figura 7.2

Funcția care generează rezultatul dorit nu are intrări. Ea este definită simplu de

$$\text{Hamming} = \langle 1 \rangle^{\text{dup3}(\text{prod}_2(\text{Hamming}), \text{prod}_3(\text{Hamming}), \text{prod}_5(\text{Hamming}))}$$

Metoda funcțională pentru o rețea de procesoare este foarte diferită de cea dată în această carte în următoarele aspecte

- (1) O implementare generală necesită bufferare nemărginită pe toate canalele.
- (2) Fiecare valoare introdusă în buffer trebuie să fie reținută în buffer până ce toate procesele receptoare l-au preluat, ceea ce poate fi făcut la momente diferite de timp.
- (3) Nu există posibilitatea pentru un proces de a aștepta una din două intrări, oricare ar sosi prima.
- (4) Procesele sunt toate deterministe.

Cercetări recente au dus la reducerea ineficienței condițiilor (1) și (2) și la relaxarea restricțiilor (3) și (4).

7.3.4 Comunicație nebufferată

De mulți ani, eu am decis să aleg comunicația nebufferată (sincronizată) ca un lucru de bază. Motivele au fost

- (1) Corespunde îndeaproape conexiunilor directe ale procesoarelor (prin fire). Firele nu pot reține mesaje.
- (2) Corespunde îndeaproape efectului de apel și întoarcere din subrutină pe un singur procesor, prin copierea valorilor parametrilor și rezultatelor.

- (3) Când se dorește bufferarea, poate fi implementată simplu ca un proces, iar gradul de bufferare poate fi precis controlat de programator.
- (4) Alte dezavantaje ale bufferelor au fost menționate la sfârșitul paragrafului 7.3.2.

Desigur, nici unul din aceste argumente nu este pe deplin convingător. De exemplu, dacă comunicarea bufferată ar fi fost luată ca primitivă nu s-ar fi putut face nici o diferență logică în cazul banal al alegerii între un apel și un return dintr-o subrutină. Sincronizarea poate fi realizată în toate celelalte cazuri prin succedarea fiecărei ieșiri cu recepția unei confirmări, iar a oricărei intrări prin emiterea unei confirmări.

7.3.5 Procese secvențiale comunicante

Acesta a fost titlul primei mele expuneri despre un limbaj de programare bazat pe concurență și comunicare. Respectiva propunere diferă de această carte în două aspecte semnificative.

(1) Compunere paralelă

Canalele nu sunt numite. În loc de numire, procesele componente ale unei construcții paralele au nume unice, prefixate prin ::

$$*[a::P||b::Q|| \dots ||c::R]$$

În procesul P comanda $b!v$ generează valoarea v procesului etichetat b . Această valoare este recepționată printr-o comandă $a?x$ intervenind în procesul Q . Numele proceselor sunt locale comenzi paralele în care sunt introduse și comunicațiile între procesele concurente sunt mascate.

Avantajul acestei scheme rezidă în faptul că nu mai este nevoie de a introduce în limbaj vreun concept de canal sau o declarație de canal. Mai mult, este logic imposibil de violat restricția ca un canal să existe între mai mult de două procese iar unul dintre ele să-l utilizeze pentru intrare și altul pentru ieșire. Dar sunt și câteva dezavantaje atât în practică cât și în teorie.

- (1) Un dezavantaj practic serios este acela că un proces subordonat trebuie să știe numele procesului care-l utilizează, aceasta complicând construcția bibliotecii de procese subordonate.
- (2) Un dezavantaj din punct de vedere matematic este că operatorul de compunere paralelă are un număr variabil de parametri și nu poate fi redus la un operator binar asociativ, ca \parallel .

(2) Terminare automată

În prima versiune, toate procesele dintr-o compunere paralelă trebuiau să se termine. Motivul consta în speranța ca corectitudinea unui proces să poată fi specificată în același fel ca *post-condiția* pentru un program convențional, cu alte cuvinte printr-un predicat adevărat la terminarea cu succes. (Speranța nu s-a împlinit și alte metode de demonstrare (paragraful 1.10) par acum mai satisfăcătoare). Obligația ca un proces subordonat să se termine determină obligația deranjantă pentru procesul care-l utilizează de a semnala terminarea sa tuturor subordonaților. De aceea s-a introdus o convenție *ad-hoc*. O buclă de forma

$$*[a?x \rightarrow P \square b?x \rightarrow Q \square \dots]$$

se termină automat la terminarea tuturor proceselor conectate la a, b, \dots de la care se cere un mesaj de intrare. Aceasta permite procesului subordonat de a termina orice secvență de cod necesară înaintea terminării propriu-zise, o caracteristică care s-a dovedit utilă în SIMULA și PASCAL PLUS.

Problema cu această convenție constă în complexitatea definirii și implementării. Metodele de a demonstra corectitudinea unui program nu par mai simple cu ea decât fără. Acum îmi apare mai normal (ca în paragraful 4.5) de a relaxa restricția ca un proces subordonat să trebuiască să se termine, în cazuri mai complicate trebuind să fie luate alt gen de măsuri (paragraful 6.4).

7.3.6 Occam

În contrast cu Ada, Occam este un limbaj de programare foarte simplu și care urmărește îndeaproape principiile expuse în această carte. Cele mai evidente și contrastante deosebiri sunt notaționale. Sintaxa Occam este proiectată pentru a fi compusă direct pe ecran cu ajutorul unui editor conector de sintaxă, folosind operatori prefix în loc de sufix și indentarea în locul parantezelor.

$\begin{array}{l} SEQ \\ \quad P \\ \quad Q \\ \quad R \end{array}$	pentru $(P;Q;R)$
---	------------------

$\begin{array}{l} PAR \\ \quad P \\ \quad Q \\ \quad R \end{array}$	pentru $(P Q R)$
---	--------------------

<i>ALT</i>	
<i>c?x</i>	pentru ($c?x \rightarrow P \sqcap d?y \rightarrow Q$)
<i>P</i>	
<i>d?y</i>	
<i>Q</i>	
<i>IF</i>	
<i>B</i>	pentru ($P \triangleleft B \triangleright Q$)
<i>P</i>	
<i>NOT B</i>	
<i>Q</i>	
<i>WHILE B</i>	pentru ($B * P$)
<i>P</i>	

Construcția *ALT* corespunde declarației *select* din Ada și oferă o gamă similară de opțiuni. Selecția unei alternative poate fi inhibată prin falsitatea unei condiții boolene *B*

B & c?x
P

Această intrare poate fi înlocuită printr-un *SKIP*, caz în care alternativa poate fi selectată ori de câte ori garda booleană este adevărată sau poate fi înlocuită printr-o așteptare care permite alternativei să fie selectată după scurgerea unui interval specific. Limbajul Occam nu are notații distincte pentru conduite (paragraful 4.4), procese subordonate (paragraful 4.5) sau procese partajate (paragraful 6.4). Toate situațiile de comunicație trebuie realizate prin identitatea explicită a numelor canalelor. Pentru a veni în ajutor, procedurile pot fi declarate cu parametri canale. Procesul simplu de copiere poate fi declarat

```
PROC copie(CHAN stg,dr) =
  WHILE TRUE
    VAR x:
    SEQ
      stg?x
      dr!x;
```

Bufferul dublu *COPIE*→*COPIE* poate fi acum construit

```
CHAN mij:
PAR
  copie(stg,mij)
  copie(mij,dr)
```

Un lanț de n buffere poate fi construit utilizând un masiv de n canale și o formă iterativă de construcție paralelă care construiește $n-2$ procese, una pentru fiecare valoare a lui i între 0 și $n-3$ inclusiv

```
CHAN mij[n-1]:
PAR
    copie(stg,mij[0])
    PAR i=[0 FOR n-2]
        copie (mij[i], mij[i+1] )
    copie(mij[n-2],dr)
```

Din cauză că Occam se intenționează a fi implementat cu alocare statică de memorie la un număr fixat de procesoare, valoarea lui n în exemplul de mai sus trebuie să fie o constantă. Din același motiv, procedurile recursive nu sunt permise.

O construcție similară poate fi utilizată pentru a realiza efectul unor procese subordonate, de exemplu

```
PROC dublu(stg,dr)=
    WHILE TRUE
        VAR x.
        SEQ
            stg?x
            dr!(x+x):
```

Această construcție poate fi declarată subordonată unui singur proces P care o utilizează

```
CHAN dub.stg,dub.dr:
PAR
    dublu(dub.stg,dub.dr)
P
```

În P un număr este dublat prin

```
dub.stg!4;dub.dr?y; ...
```

Procese pot fi partajate utilizând masive de canale (cu câte un element pentru fiecare proces utilizator) și o formă iterativă a construcției *ALT*. De exemplu, să luăm un integrator care după fiecare nouă recepție transmite suma tuturor elementelor pe care le-a recepționat până atunci

CHAN *adună*[*n*], *integral*[*n*]:

PAR

VAR *suma*, *x*:

SEQ

suma:=0

ALT *i*:=*[0 FOR n]*

adună[*i*]?*x*

SEQ

suma:=*suma*+*x*

integral[*i*]:=*suma*

PAR *i*:=*[0 FOR n]*

 ...*procese utilizator*...

Ca și în Ada, Occam permite programatorului asignarea de priorități relative unor procese compuse în paralel. Acesta se face utilizând *PRI PAR* în loc de *PAR*. Procesele din capul listei au priorități mai mari. Facilitățile de editare pe ecran asigurate de limbaj permit reorganizarea proceselor când e necesar. O opțiune similară este oferită pentru construcția *ALT* de *PRI ALT*. Aceasta permite pentru mai mult de o alternativă gata pentru selecție imediată, alegerea primei textual apărute – altfel efectul este același ca și pentru *ALT*. Desigur, programatorul este rugat să se asigure că programele sale sunt corecte logic, independent de asigurarea priorităților.

Există de asemenea facilități pentru distribuirea proceselor între procesoare distincte și pentru specificarea pinilor fizici de la fiecare procesor ce vor fi folosiți pentru fiecare canal relevant din programul Occam precum și pinilor folosiți pentru încărcarea codului programului însuși.

7.4 Modele matematice

Recunoașterea ideii că un limbaj de programare ar trebui să aibă o semnificație matematică precisă sau semantică datează de la începuturile lui 1960. Aspectul matematic asigură o specificare sigură, fără ambiguități, precisă și stabilă a limbajului pentru a servi ca interfață agreabilă între utilizatori și implementatori. Mai mult, de aici derivă motivele sigure pentru dezideratul ca implementări diferite să fie implementări ale aceluiași limbaj. Astfel, semanticile matematice sunt la fel de importante pentru obiectivul de standardizare a limbajului ca și măsurătorile și calculele la standardizarea șuruburilor și piulițelor.

După 1960 a fost recunoscut rolul important al semanticilor matematice și anume acela de a ajuta programatorul să se debaraseze de obligația stabilirii corectitudinii programului său. Într-adevăr, R.W. Floyd a sugerat ca semanticile să fie formulate mai bine ca o mulțime de reguli de demonstrare

(axiome) decât ca un model matematic explicit. Această sugestie a fost adoptată în specificarea limbajelor PASCAL, Euclid și Gypsy.

Prima proiectare a Proceselor Secvențiale Comunicante (paragraful 7.3.5) nu avea o semantică matematică și lăsa deschise un număr de întrebări de proiectare ca de exemplu :

- (1) Este posibil de a imbrica o comandă paralelă în alta ?
- (2) Dacă da, este posibil de a scrie o procedură recursivă care se apelează în paralel ?
- (3) Este teoretic posibil de a folosi comenzi de ieșire în gărzi ?

Modelul matematic dat în această carte răspunde cu "Da" la toate aceste întrebări.

7.4.1 Calculul sistemelor comunicante

O deschidere majoră în modelarea matematică a concurenței a fost făcută de Robin Milner. Obiectivul major al investigației sale a fost asigurarea unui cadru pentru construcția și compararea diferitelor modele la diferite nivele de abstractizare. Astfel, el a început cu sintaxa de bază a expresiilor dedicate să semnifice procese și a definit o serie de echivalențe între expresii din care cele mai importante sunt :

- echivalență tare
- echivalență observațională
- congruență observațională

Fiecare echivalență definește un model diferit de concurență. Inițialele CCS se referă de obicei la modelul obținut prin adoptarea congruenței observaționale ca definiție a egalității între procese.

Notațiile de bază din CCS sunt ilustrate prin corespondența următoare

$a.P$	corespunde la $a \rightarrow P$
$(a.P) + (b.Q)$	corespunde la $(a \rightarrow P b \rightarrow Q)$
NIL	corespunde la $STOP$

Mai importante decât aceste diferențe notaționale sunt distincțiile în tratarea mascării. În CCS există un simbol special τ care semnifică apariția unui eveniment mascat sau o tranziție internă. Avantajul păstrării acestei înregistrări trecute a unui eveniment mascat este acela că poate fi utilizat fără restricții în gărzile ecuațiilor recursive asigurându-ne astfel că au soluții unice, așa cum s-a arătat în paragraful 2.8.3. Un al doilea (dar poate mai puțin semnificativ) avantaj este acela că procesele care se angajează într-o secvență

nemărginită de τ -uri nu se reduc toate la *CHAOS*. Astfel, distincții utile pot fi făcute între procese divergente. Totuși CCS eșuează în distingerea unui proces posibil divergent de unul care este similar în comportare dar nu-i divergent. Este de așteptat ca aceasta să determine imposibilitatea unei implementări eficiente pentru limbajul CCS.

CCS nu include \sqcap ca operator primitiv. Totuși, nedeterminismul poate fi implementat prin utilizarea lui τ , de exemplu

$$(\tau.P) + (\tau.Q)$$

$$(\tau.P) + (a.Q)$$

corespunde la $(P \sqcap Q)$

corespunde la $P \sqcap (P \sqcap (a \rightarrow Q))$

Dar aceste corespondențe nu sunt exacte din cauză că în CCS nedeterminismul definit de τ nu este asociativ așa cum rezultă din faptul că arborii din fig. 7.3. sunt distincți.

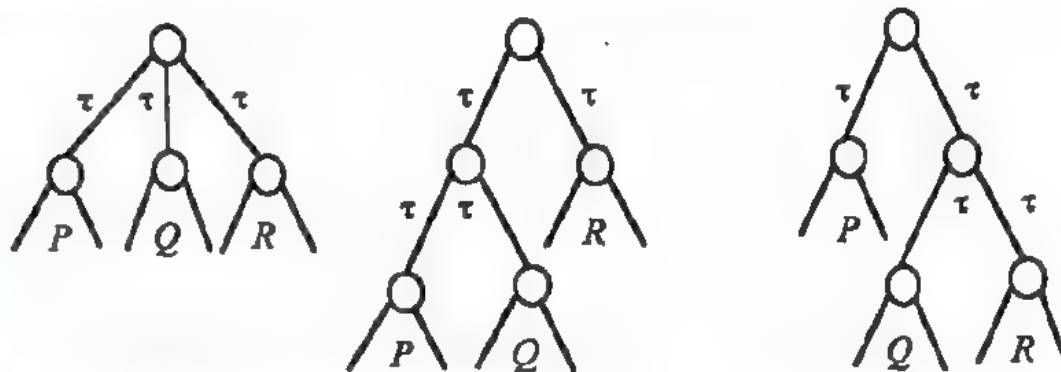


Figura 7.3

Mai mult, prefixarea nu se distribuie cu nedeterminismul, din cauză că arborii din fig. 7.4 sunt distincți când $P \neq Q$

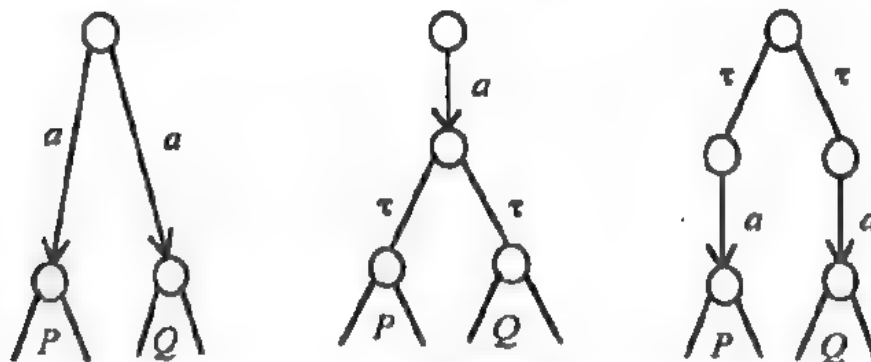


Figura 7.4

Aceste exemple ne arată că CCS face multe distincții între procese care ar putea fi privite ca identice în această carte. Motivul pentru aceasta este că CCS se intenționează a fi un cadru pentru o familie de modele, fiecare din ele putând face mai multe identificări decât CCS, dar nu mai puține. Pentru a

evita restrângerea gamei de modele, CCS face numai acele identificări care par a fi absolut necesare. În modelul matematic din această carte am urmărit exact scopul opus – am făcut cât mai multe identificări posibile, păstrând numai distincțiile cele mai importante. De aceea avem o mulțime mult mai mare de legi algebrice. Se speră că aceste legi vor fi de utilitate practică în raționamente de proiectare și în implementări. În particular ele permit mai multe transformări și optimizări decât CCS.

Operatorul de concurență de bază din CCS se notează prin $|$. El este cu mult mai complicat decât operatorul \parallel , în el incluzându-se aspecte de mascare, nedeterminism, întretesere ca și sincronizare. Fiecare eveniment din CCS are două forme, simplă (a) sau barată (\bar{a}). Când două procese sunt reunite pentru a evolua concurrent, sincronizarea apare numai când un proces se angajează în evenimentul cu bară și celălalt proces se angajează în evenimentul corespunzător simplu. Participarea lor comună la un astfel de eveniment este mascată prin conversia imediată la τ . Totuși, sincronizarea nu este obligatorie. Fiecare din cele două evenimente poate apare vizibil și independent ca o interacțiune cu mediul extern. Astfel în CCS

$$\begin{aligned}(a.P)|(b.Q) &= a.(P|(b.Q)) + b.((a.P)|Q) \\ (a.P)|(\bar{a}.Q) &= a.(P|(\bar{a}.Q)) + \bar{a}.((a.P)|Q) \\ (a.P)|(\bar{a}.Q) &= \tau.(P|Q) + a.(P|(\bar{a}.Q)) + \bar{a}.((a.P)|Q)\end{aligned}$$

Mai mult, numai două procese se pot angaja într-un eveniment de sincronizare. Dacă mai mult decât două procese sunt gata, alegerea unei perechi este nedeterministă

$$\begin{aligned}(a.P)|(a.Q)|(\bar{a}.R) &= \tau.(P|(a.Q)|R) + \tau.((a.P)|Q|R) \\ &\quad + a.(P|(a.Q)|(\bar{a}.R)) \\ &\quad + \bar{a}.((a.P)|Q|(\bar{a}.R)) \\ &\quad + \bar{a}.((a.P)|(a.Q)|R)\end{aligned}$$

Din cauza complexității operatorului paralel nu este nevoie de un operator de mascare. În locul lui există un operator de *restricție* \backslash , care împiedică pur și simplu aparițiile tuturor evenimentelor relevante și le elimină din alfabetul procesului, împreună cu varianta lor cu bară. Efectul este ilustrat de următoarele legi din CCS

$$\begin{aligned}(a.P)\backslash\{a\} &= (\bar{a}.P)\backslash\{a\} = NIL \\ (P+Q)\backslash\{a\} &= (P\backslash\{a\}) + (Q\backslash\{a\}) \\ ((a.P)|(a.Q))\backslash\{a\} &= \tau.((P|Q)\backslash\{a\})\end{aligned}$$

$$((a.P)|(a.Q)|(\overline{a}.R)) \setminus \{a\} = \tau.((P|(a.Q)|R) \setminus \{a\}) \\ + \tau.(((a.P)|Q|R) \setminus \{a\})$$

Ultima lege de mai sus ilustrează puterea operatorului paralel din CCS în realizarea efectului de partajare a procesului $(\overline{a}.R)$ între două procese utilizator $(a.P)$ și $(a.Q)$. Unul din obiectivele CCS-ului a fost de a realiza maximum de putere de expresie cu cât mai puțini operatori distincți posibili. Aceasta este sursa eleganței și puterii CCS-ului și într-adevăr simplifică investigația familiilor de modele definite de diferite relații de echivalență.

În această carte s-a ales o cale complementară. Simplitatea este văzută prin proiectarea unui model simplu, în termenii căruia este ușor de definit cât mai mulți operatori potriviți pentru investigarea unei game de concepte distincte. De exemplu, alternativa nedeterministă \sqcap introduce nedeterminismul în forma sa cea mai pură și este cât se poate de independentă de controlul mediului reprezentat prin $(x.B \rightarrow P(x))$. Similar, \parallel introduce concurența și sincronizarea destul de independent de nedeterminism sau mascare, fiecare dintre ele fiind reprezentat de un operator distinct. Faptul că aceste concepte sunt distincte este poate indicat de simplitatea legilor algebrice. În aplicarea practică a teoriilor matematice utile pare a fi necesară o gamă cât mai largă și rezonabilă de operatori. Minimizarea mulțimii operatorilor este de asemenea utilă, mai ales în investigațiile teoretice.

Milner a introdus o formă de logică modală pentru a specifica comportarea observabilă a unui proces. Modalitatea

$$\diamond a S$$

descrie un proces care poate efectua a și apoi să se comporte așa cum este descris de S iar dualul său

$$\Box a S$$

descrie un proces care dacă pornește cu a trebuie să se comporte la fel ca S . Este definit un calcul de corectitudine care permite dovedirea faptului că un proces P satisface specificația S , un fapt care este exprimat în notația logică tradițională

$$P \models S$$

Calculul este foarte diferit de acela din relația sat, din cauză că se bazează mai mult pe structura specificării decât pe structura programelor. De exemplu, regula pentru negație este

Dacă *nu* este adevărat că $P \models F$
atunci $P \models \neg F$

Aceasta înseamnă că tot procesul P trebuie scris înaintea pornirii demonstrării corectitudinii sale. În contrast, utilizarea lui *sat* permite ca demonstrarea corectitudinii unui proces component să fie construită din demonstrarea corectitudinii părților sale. Logica modală este un subiect de mare interes teoretic, dar în contextul proceselor comunicante nu a arătat încă mare lucru pentru o aplicare utilă.

În general, egalitatea în CCS este o relație tare deoarece procese egale trebuie să semene între ele atât în comportarea lor observabilă cât și în structura comportării lor mascate. De aceea CCS este un model bun pentru formularea și explorarea variațiilor definiției slabe de echivalență care ignoră diversele aspecte ale comportării ascunse. Milner a realizat aceasta prin introducerea conceptului de echivalență observațională. Aceasta implică definirea unei mulțimi de observații sau evenimente (experimente) care pot fi făcute cu un proces. Două procese sunt echivalente dacă nu există nici o observație care poate fi făcută pentru unul și nu poate fi făcută pentru celălalt – o frumoasă aplicare a principiului filozofic de identitate a indescifrabilului. Principiul a fost luat ca bază a teoriei matematice din carte, care echivalează un proces cu mulțimea observațiilor care pot fi făcute asupra comportării sale. Un semn al succesului principiului este acela că două procese P și Q sunt echivalente dacă și numai dacă ele satisfac aceeași specificație

$$\forall S, P \models S \iff Q \models S$$

Din nefericire, nu totdeauna este la fel de simplu. Dacă două procese trebuie privite ca egale, rezultatul transformării lor prin aceeași funcție ar trebui să fie de asemenea egal.

$$(P=Q) \Rightarrow (F(P)=F(Q))$$

Dacă τ se presupune că este mascat, o definiție naturală a unei observații ar putea conduce la echivalența

$$(\tau.P) = P$$

Totuși $(\tau.P + \tau.NIL)$ n-ar trebui să fie echivalent cu $(P + NIL)$, care este egal cu P , deoarece primul poate face o alegere nedeterministă către blocaj în loc să se comporte ca P .

Soluția lui Milner la această problemă este utilizarea *congruenței* în locul echivalenței. Într-un experimente care pot fi realizate cu procesul P unul este acela de a-l așeza într-un mediu $F(P)$ unde este compusă din alte

procese cu ajutorul operatorilor limbajului) și apoi de a observa comportarea ansamblului. Procesele P și Q sunt (din punct de vedere observabil) congruente dacă pentru fiecare F exprimată în limbaj, procesul $F(P)$ este echivalent observabil cu $F(Q)$. Potrivit acestei definiții, $\tau.P$ nu este congruent cu P . Descoperirea unei mulțimi întregi de legi de congruență este o realizare matematică semnificativă.

Necesitatea unei complexități suplimentare în cazul congruenței observaționale se datorează imposibilității de a face observații suficient de penetrante asupra comportării lui P fără plasarea lui într-un mediu $F(P)$. De aceea, noi am fost nevoiți să introducem conceptul de *mulțime* de refuz față de refuzul unui singur eveniment. Mulțimea de refuz pare a fi cel mai slab fel de observație care reprezintă eficient posibilitatea blocajului nedeterminist, de aceea ne conduce la o echivalență mai slabă și la o mulțime mult mai puternică de legi decât în CCS.

Descrierea dată mai sus a subliniat pe larg diferențele față de CCS și a stabilit încă o dată spectrul aplicațiilor practice ale metodei prezentată în această carte. Cele două metode au în comun o caracteristică foarte importantă și anume o bază matematică promițătoare pentru raționamente despre specificații, modelare și implementare. Fiecare din ele poate fi utilizată atât pentru investigații teoretice cât și pentru aplicații practice.



Bibliografie selectivă

- Conway, M.E. "Design of a Separable Transition-Diagram Compiler", *Comm. ACM* 6 (7), 8-15 (1963)
Articol clasic despre corutine
- Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", *Comm. ACM* 17 (10), 549-557 (1974)
Un concept de programare menit să ajute la construcția sistemelor de operare
- Hoare, C.A.R. "Communicating Sequential Processes", *Comm. ACM* 21 (8), 666-677 (1978)
- Un model de programare - o versiune anterioară celei prezentate în această carte
- Milner, R. *A Calculus of Communicating Systems*, Lectures Notes in Computer Science 92, Springer Verlag, New-York (1980)
O tratare matematică clară a teoriei generale despre concurență și sincronizare
- Kahn, G. "The Semantics of a simple language for Parallel Programming", în *Information Processing*, 74, North Holland, Amsterdam pp.471-475 (1974)
O tratare elegantă a multiprogramării funcționale
- Welsh, J. și McKeag, R.M. *Structured System Programming*, Prentice-Hall London, pp. 324 (1980)
O trecere în revistă a lui PASCAL PLUS și a utilizării sale în structurarea unui sistem de operare și a unui compilator
- Filman, R.E. și Friedman, D.P. *Coordinated Computing, Tools and Techniques for Distributed Software*, McGraw-Hill pp. 370 (1974)
O utilă trecere în revistă cu trimiteri bibliografice
- Dahl, O-J. "Hierarchical Program Structures", în *Structured Programming*, Academic Press, London pp. 175-220 (1982)
O introducere la ideile de bază ale lui SIMULA 67
- (INMOS Ltd.) *occam™ Programming Manual* Prentice-Hall International, pp. 100 (1984)

(ANSI/MIL-STD *Ada*TM *Reference Manual*
1815A)

Capitolul 9 descrie caracteristica de tasking

Brookes, S.D., "A Theory of Communicating Sequential Processes",

Hoare, C.A.R. *Journal ACM* 31 (7), 560-599 (1984)

și Roscoe, A.W. O trecere în revistă a aspectelor matematice ale proceselor
nedeterminate, dar fără divergențe

Brookes, S.D. și "An Improved Failures Model for Communicating

Roscoe, A.W. Sequential Processes", în *Proceedings NSF-SERC
Seminar on Concurrency*, Springer Verlag, New-York,
Lectures Notes in Computer Science (1985)

Îmbunătățirea este în legătură cu divergențele

INDEX

A

<i>ACADEMIE</i> ,	77
<i>ACADEMIE_NOUĂ</i> ,	78
<i>acc</i> ,	198
<i>accept</i> ,	245
<i>Ada</i> ,	244
<i>alfabet</i> ,	21
<i>alegere</i> ,	29
<i>ALGOL 60</i> ,	240
<i>alternarea</i> ,	191
<i>alternativă</i> ,	30
<i>alegere2</i> ,	37
<i>alegere binară</i> ,	108
<i>alegere independentă</i> ,	105
<i>alegere multiplă</i> ,	106
<i>apel de procedură la distanță</i> ,	217
<i>apel de procedură local</i> ,	217
<i>apoi</i> ,	23
<i>arbore binar</i> ,	172
<i>arce</i> ,	32
<i>ARPAnet</i> ,	248
<i>asignare</i> ,	195
<i>AVC</i> ,	28
<i>AVCB</i> ,	28
<i>AVMCCM</i> ,	29
<i>AVS</i> ,	27
<i>AVS2</i> ,	29
<i>AVSUNET</i> ,	69

B

<i>backtracking</i> ,	182
<i>bloc de control</i> ,	223
<i>blocaj</i> ,	66
<i>blocaj prin ciclare infinită</i> ,	161
<i>bucă</i> ,	180

<i>BUFFER</i> ,	142
<i>buffer</i> ,	164
<i>buffer dublu</i> ,	159

C

<i>cale</i> ,	51
<i>canal</i> ,	137
<i>canal de intrare</i> ,	138
<i>canal de ieșire</i> ,	138
<i>cap</i> ,	43
<i>cartelă separatoare</i> ,	227
<i>catastrofă</i> ,	190
<i>CCS</i> ,	256
<i>cea mai mică margine</i>	
<i>superioară</i> ,	95
<i>CEAS</i> ,	25
<i>CHAOS</i> ,	130
<i>ciclicitatea</i> ,	53
<i>cititor de cartele partajat</i> ,	227
<i>CÎT</i> ,	196
<i>CÎTLUNG</i> ,	197
<i>coadă de ieșire</i> ,	229
<i>coadă de intrare</i> ,	229
<i>cobegin</i> ,	237
<i>codarea în fază</i> ,	166
<i>coend</i> ,	237
<i>compilare separată</i> ,	246
<i>compunerea</i> ,	57
<i>compunere secvențială</i> ,	183
<i>comunicare</i> ,	137
<i>comutare de pachete</i> ,	238
<i>comutator</i> ,	225
<i>concatenare</i> ,	41
<i>concurență</i> ,	70

condiție, 195
 conducte, 156
 confirmare, 170
 congruență, 260
 constructiv, 98
 continuă, 96
 controlul fluxului, 168
 COPIE, 139
 COPIEBIT, 29
 cu gardă, 26
 cu gardă stîngă, 162

D

deplasare stîngă, 146
 desfășurare, 25
 DESPACHET, 140
 dezacord de structură, 158
 diagrame de conexiune, 74
 disjunct, 238
 distributiv, 41, 105
 diverge, 119
 divergență, 117
 domeniu, 133
 DUBLU, 139
 după, 51

E

echivalență, 259
 eliberează, 210
 estemembru, 46
 esteprefix, 47
 esteurmă, 51
 eşecuri, 133
 elich, 89
 etichetare multiplă, 89
 etichetarea proceselor, 85
 Euclid, 256
 evaluare progresivă, 39
 eveniment, 21
 execuție simbolică, 159
 expresii regulate, 182
 extensie de alfabet, 116

F

factorial, 171
 FCFS, 232
 FIB, 146
 FIFO, 232
 FIL, 75
 fișier de lucru, 220
 FIRE, 165
 fork, 237
 FURC, 76

G

gramatică liberă de context, 182
 Gypsy, 256

H

Hamming, 249

I

implementare, 36
 indexare, 56
 instance, 241
 instrucțiuni incluse, 241
 interacțiune, 65
 interferență, 213
 interuptibil, 188
 invers, 52

Î

ÎMPACHET, 140
 înlănțuire, 157
 întreruperi, 188
 întrețesere, 55, 122

J

join, 237

L

LABEL, 37
 lanț, 95
 LAN72, 82
 legare statică, 39

legi,	33
legi algebrice,	109
limbaj,	180
limită,	95
LISP,	38
LISPkit,	38
lungime,	45

M

marker de sfârșit de fișier,	220
masa filozofilor,	75
<i>masc</i> ,	118
mascare,	113
masiv sistolic,	151
master,	168
mediu de memorare,	218
memorie partajată,	213
meniu,	30
<i>meniu</i> ,	38
<i>mesaj</i> ,	137
<i>MM_n</i> ,	32
modul sistem,	226
modularitate,	158,231
monitor,	240
monitoare imbricate,	242
monotonă,	45
multiplicarea firelor de execuție,	236
multiprocesare funcțională,	248
multiprocesor,	215
<i>MULTIME</i> ,	171
mulțime finită,	171
mulțimi de <i>gata</i> ,	111

N

nedestructivă,	99
nedeterminism angelic,	107
<i>NIL</i> ,	46
noduri,	32
<i>NONSTOP</i> ,	126
nume de intrare,	245

O

occam,	252
<i>ocupă</i> ,	211
ordine parțială,	44,95
ordine parțială completă,	95
organizare în coadă,	228

P

partajarea timpului,	236
PASCAL PLUS,	240,243
<i>PIDGINGOL</i> ,	180
planificare,	231
pointeri,	223
postcondiție,	252
pragma,	246
precondiție	200
prefix,	23,39,44
preluare infinită,	80,232
prioritate,	246
procedură,	217
proces,	22
proces determinist,	93
proces principal,	168
procesare batch,	226
procesare distribuită,	215
procese acceptoare,	181
produs scalar,	151
programare secvențială,	195
propoziție,	180
protecție,	242
protocol,	147
punct fix,	95
puncte de control,	192
puncte de control multiple,	193

R

RC4000,	248
receptor,	165
recursivitate,	25
recursivitate fără gardă,	117
recursivitate mutuală,	31
refuzuri,	110

regiune critică,	215,238
regiuni critice condiționale,	238
regulă de copiere,	243
rendezvous,	245
reprezentări grafice,	32,138
resursă reală,	242
resursă virtuală,	223
reutilizabil serial,	215
rețele de fluxuri de date,	149
rezervare a biletelor de avion,	209

S

<i>sau1</i> ,	107
<i>sau2</i> ,	107
<i>sau3</i> ,	107
schimbare de simbol,	80
sector,	218
selecție,	56
semafor,	214
semafor de excludere binar,	214
semantici,	255
SIMULA 67,	240
sincronizare,	66
singleton,	42
sistem de baze de date,	192
sistem de operare,	208
SKIP,	179
slave,	168
specificare,	58
stare,	31
stea,	43,54
stivă,	142
STOP _A ,	23
straturi,	165
strictă,	41
structură de date,	171
structură de date partajate,	209
subordonare,	168
subordonat,	168
subrutină,	169
subrutină reentrantă,	216
succes,	179
sumă ponderată,	150

T

terminare cu succes,	57
transmițător,	165
transparent,	147
tranziție,	32

U

UNIFICĂ,	141
unicitatea soluțiilor,	97
UNIX,	248
<i>ur</i> ,	59
urmă,	39,72

V

valet,	78
valet tânăr,	89
VAL _T ,	89
VAL _{T_M} ,	89
<i>var</i> ,	198
VAR,	141
VARB,	86
variabile libere,	58